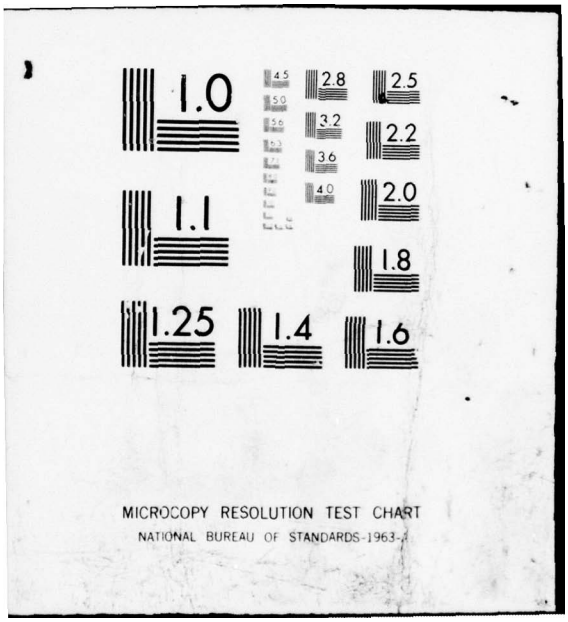
AD-A064 910 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
SMITE INSTALLATION AND ANALYSIS.(U)
DEC 78 B PRESS, L A PRENTICE F30602-77-C-0089
UNCLASSIFIED TRW-30417-6003-RU-00 RADC-TR-78-212 NL

1 OF 2
AD
A064910


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12) LEVEL II

RADC-TR-78-212
Final Technical Report
December 1978

SMITE INSTALLATION & ANALYSIS

TRW

B. Press
L. A. Prentice

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-78-212 has been reviewed and is approved for publication.

APPROVED: Frederick A. Normand

FREDERICK A. NORMAND
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization please notify RADC (ISCA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-78-212	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): SMITE INSTALLATION AND ANALYSIS	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report, May 77 - May 78
6. PERFORMING ORG. REPORT NUMBER: 30417-6003-RU-00	
7. AUTHOR(s): B. Press, L.A. Prentice	8. CONTRACT OR GRANT NUMBER(s): F30602-77-C-0089
9. PERFORMING ORGANIZATION NAME AND ADDRESS: TRW/Defense and Space Systems Group, One Space Park, Redondo Beach CA 90278	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F, 55501701
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISCA), Griffiss AFB NY 13441	12. REPORT DATE: December 1978
	13. NUMBER OF PAGES: 99
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
103p.	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DDC RECEIVED FEB 26 1979 B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Frederick A. Normand (ISCA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Hardware Description Language
Programming
Emulation
Compiler
Simulation
Computer Architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents the findings and conclusions of research and studies into the use of extensible features within the SMITE Language. SMITE is a computer hardware description language that produces an emulation of the machine that can be loaded and run on a QM-1 computer. The extensibility feature will allow the description of non-standard architectures and architectures not yet envisioned.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Table of Contents

EVALUATION

RADC is currently building a computer emulation facility to assist in evaluation of hardware/software/firmware tradeoffs necessary in the development of system architectures under TPO 5, Thrust 5.1. As part of this effort, RADC has purchased a QM-1 microprogrammable computer which is designed to run computer emulations, and to get access to SMITE which is a Higher Order Language for describing computer architecture emulations and a compiler which produces code to emulate said architectures on the QM-1 computer. This effort also studied the possibility of being able to extend SMITE to make it a more useful hardware description language. The results of this study are being incorporated into an Advanced SMITE which is being written in a subset of PL-1 to be run on the MULTICS operating system at RADC.

Frederick A. Normand

FREDERICK A. NORMAND
Project Engineer

Cont'd on p 1473A

Preface

This report, CDRL A003, is the final technical report of contract F30602-77-C-0089, which addressed the the following technical items:

1. Installation of the SMITE compiler.

2. Installation of the SMITE Applications Support Software (SASS).

3. Training of RADC personnel in the use of the SMITE language.

4. Analysis of the use of extensibility features within SMITE including use of technology derived from the Alphard language.

The SMITE compiler was installed on the CDC-6000 system at the Air Force Weapons Laboratory (AFWL), Kirtland Air Force Base, New Mexico. This allows access to the compiler by RADC personnel through the use of the ARPANET. SASS was installed on the Nanodata QM-1 at the Rome Air Development Center (RADC), Rome, New York to support RADC use of emulations produced by the compiler. The test plan and procedures used for both of these installations are contained in CDRL A004, SMITE INSTALLATION & ANALYSIS - TEST PLAN AND PROCEDURES, of this contract.

Training of RADC personnel was accomplished in a one week course at RADC during the contract. As part of the training preparation CDRL A005, SMITE INSTALLATION & ANALYSIS - SMITE TRAINING MANUAL, was published.

This report presents the findings of the extensibility analysis.

1.0 Introduction

1.1 Summary

This report presents the findings and conclusions of research and studies into the use of extensible features within computer description language technology. Readers of this report are assumed familiar with SMITE computer description language [1] and the Nanodata QM-1 Computer [2].

The value of SMITE as a computer description language and emulation development tool can be significantly enhanced by providing extensibility features within the language. With addition of extensibility, non-standard architectures can be described and emulated. Also, the descriptions of existing architectures for which SMITE is not quite suited (such as various forms of arithmetic other than 2's complement) will be enhanced. Extensibility will also allow SMITE to be used for architectures not yet envisioned.

These studies address two areas of extensibility:

1. Downward extensibility which gives the user access to the Multi language and QM-1 system functions.

2. Upward extensibility which allows the user to modify the existing language to define a new language that corresponds to his specific needs.

The downward extensibility features studied provide the ability to produce more efficient emulations by allowing the user to force into the microcode level those functions which are performed repeatedly or are not easily described in SMITE. These features allow the user to code directly in MULTI and also allow the ability to define new MULTI instructions the compiler did not initially recognize.

The upward extensibility features studied allow the user to "tailor" the language to his own needs. Using this ability the user can add new language constructs to aid in the representation of the problem at hand. This mechanism will also provide the user with more refined methods of abstraction than current SMITE allows. Complete and modularized descriptions must be composed of several levels of complexity, such as functional, behavioral, and structural. To provide these levels it is necessary to have a high degree of abstractive ability in a language.

This report is divided into several sections containing the following information:

1. Overview and summary of the extensibility features to be added to SMITE as determined by this study (Section 2.0).

2. Working notes produced as the studies of the various extensibility areas progressed. These are not formal descriptions of the features but merely a compilation of some of the notes produced during the study to give an indication of the thought processes involved. (Sections 1.0, 2.0, and 3.0 of Appendix A, corresponding to the areas of the overall extensibility mechanism, syntax macros, and direct code, respectively).

3. The semantics and syntax of the extensible features to actually be added to the SMITE language (section 3.0).

4. SMITE Compiler Implementation notes produced during the study (Section 4.0).

5. SASS requirement notes produced during the study (Section 5.0).

6. Conclusions (Section 6.0).

1.2 References

1. TRW Defense and Space Systems Group, SMITE Reference Manual, RADC-TR-77-364, November 1977.

2. Nanodata Corporation, QM-1 Hardware Level User's Manual, March 1976.

3. Livingston, S. H., History of Manchester Computers, University of Manchester, 1975.

4. Wulf, W. A., R. L. London, and M. Shaw, Abstraction and Verification in Alphard: Introduction to Language and Methodology, Carnegie Mellon University Department of Computer Science, 1976.

5. Bixler, D. C, SMITE Language Research Final Report, TRW IOC 6413.40-10, October 1975.

6. Shaw, M., W. A. Wulf, and R. L. London, Abstraction and

Verification in ALPHARD: Defining and Specifying Iteration and Generators, Comm. ACM 20, 8 (Aug 1977), 553-564.

7. Bixler, D. C, CPDL Requirements, TRW Internal Report, May 1977.

8. Bixler, D. C, and H. M. Hart, CPDL Constructs, TRW Internal Report, July 1977.

9. Bixler, D. C, and H. M. Hart, CPDL Specification, TRW Internal Report, Sept 1977.

10. Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, 1972.

11. Early, J., An Efficient Context-Free Parsing Algorithm, Comm. ACM 13, 2 (Feb. 1970), 94-102.

12. Crocker, S. D., State Deltas: A Formalism for Representing Segments of a Computation, University of Southern California Information Sciences Institute, 1977.

13. Newcomer, J. M., Machine Independent Generation of Optimum Local Code, Carnegie Mellon University Department of Computer Science, 1975.

14. Fraser, C. W., A Knowledge-Based Code Generator Generator, ACM Sigplan Notices, August 1977, pp126-129.

1.3 Syntax Notation

Since this study describes the syntax for the extension features of SMITE, some method of language description is needed to present the information in a clear, precise fashion. The method chosen for this study is a loosely formal grammatical notation.

A formal grammar is composed in part of rules or productions. Each production specifies a textual replacement. By starting with a chosen initial symbol, which for SMITE is <smite-program>, and substituting as necessary using the productions, all legitimate forms of the language may be developed.

Three general classes of symbols appear in productions, namely terminal symbols, non-terminal symbols, and meta-linguistic symbols.

Terminal symbols are those characters and character strings which will actually appear in a SMITE program. Examples of terminal symbols include REGISTER, MEMORY, and DECLARE, and the characters ',' and ';'. Any symbol which is not a non-terminal or meta-linguistic symbol is by default a terminal symbol.

Non-terminal symbols represent abstract entities in the program, and will always be written as

<name>

where "name" is an identifier of the entity. Examples of non-terminals from SMITE include <declaration> and <processor>.

Meta-linguistic symbols are those characters used to write productions. The characters '<' and '>' are such characters, and are used to denote non-terminal symbols. The meta-linguistic symbols in addition to corner brackets in the study are interpreted as follows:

Brackets '[' and ']' are used to indicate the optional occurrence of the phrase written within them. For example:

["," <id>]

Additionally,

[<name>]*

is used to indicate that any number, including zero, of occurrences of the phrase may appear. Further,

[<name>]+

is used to indicate that one or more occurrences may appear.

The symbol '::=' is used to denote the definition of a non-terminal. All productions are of the form

<non-terminal> ::= phrase

where <non-terminal> is thereby defined to be "phrase".

The symbol '/' is used to designate alternatives. If <name> is to be derived to be A or B, the production to express this would be

<name> ::= A / B

The symbol '"' is used to indicate that a meta-linguistic symbol or other symbol that could cause confusion is being used as a terminal symbol. Thus

"["

is the terminal symbol '[' , and not the beginning of an optional or repeated phrase.

Parentheses, '(' and ')', are used to group several symbols into one logical entity to avoid confusion or to reorder evaluation of meta-symbols in the production. For example:

<name> ::= (SAM / PETE) <type>

There is no hierarchy of meta-symbols. Productions are evaluated from left to right with symbols within parentheses evaluated first.

2.0 Overview and Summary

Computer description languages provide concise, unambiguous, specific descriptions of the operation of digital devices. The functional modularity made possible by MSI, LSI, and VLSI implies the need for a corresponding descriptive modularity, whereby the devices composing a digital system may be described along with the structural information defining their interconnections. This modularity must be provided, however, without losing sight of the functionality of the complete system. Contemporary computer description languages fail to satisfy this requirement. Specification of modularity and structure is difficult, and obscures the overall system view. Specification of distributed systems is virtually impossible.

The addition of the ability to specify abstract devices, similar to the specification of abstractions or virtual machines in software, provides a solution to this descriptive problem.

To illustrate the use of the device abstraction and provide examples of the syntax for defining devices a description of a simple computer is first presented in basic SMITE. This computer is the University of Manchester Mark 1 [3] and the description begins with the declaration of the memory, registers and subregisters used within the computer. Following the declarations is a description of the instruction fetch, decode and execute cycle.

```
MARK-1: PROCESSOR;
   DECLARE
      M[0:8191]<0:31> MEMORY,
      PI<0:15> REGISTER,
         F<0:2> DEFINED PI<0:2>,
         S<0:12> DEFINED PI<3:15>,
      CR<0:12> REGISTER,
      ACC<0:31> REGISTER;
   DECLARE
      STOP EXTERNAL;
   DO FOREVER;
      BEGIN;
         PI <- M[CR]<0:15>;
         CASE F;
         '' 0: branch ''
            CR <- M[S]<19:31>;
```

```
         '' 1: branch relative ''
            CR <- CR + M[S]<19:31>;
         '' 2: load negation ''
            ACC <- - M[S];
         '' 3: store ''
            M[S] <- ACC;
         '' 4,5: subtract ''
            ACC <- ACC - M[S];
            ACC <- ACC - M[S];
         '' 6: skip if negative ''
            IF SE(ACC) < 0
               THEN CR <- CR + 1;
               END IF;
         '' 7: halt ''
            STOP;
         END CASE;
      CR <- CR + 1;
      END;
   MARK1: END;
```

2.1 Abstraction Concepts

A dichotomy exists in the use of single level computer description languages. If a description is written to accurately reflect the mechanization of a computer, the functionality may be completely obscured. Conversely, writing the description to clearly define the functionality may distort or completely suppress description of the implementation. For example, the following excerpt is the description of the floating point multiplier from a description of the Raytheon Fault Tolerant Spaceborne Computer:

```
   FLOATING-MULTIPLY:  PROCESSOR;
      FLOATING-PREP;
      REG-OP <- SLL(REG-OP, 8);
      MULTIPLY;
      IF REG-OP = 0
         THEN REG-OP <- X'80';
         ELSE BEGIN;
            REG-OP-EXP <- REG-OP-EXP + OPERAND-EXP;
            DBL-NORMALIZE;
            END;
         END IF;
      FLOATING-MULTIPLY:  END;
```

The description was written to obtain a bit-accurate emulation of the computer via compilation of the SMITE computer description to microcode, and therefore the description is strongly oriented towards a representation of the computer implementation. The

functionality of performing a floating point multiplication is obscured in the process.

The abstraction concepts proposed by Wulf, London, and Shaw in Alphard [4] provide a resolution of the functionality/implementation dichotomy by supporting the expression of both abstraction and concrete realization. In the context of software, Wulf et al. state:

> "A key concept in structured programming is abstraction: the retention of the essential properties of an object and the corollary neglect of inessential details. ... Abstraction is important to structuring programming precisely because it permits a programmer to ignore inessential detail and thereby reduce the apparent complexity of his task."

In our terms, abstraction permits the user of a device to ignore the inessential details of its implementation, using instead the specified functionality of the device.

The module of abstraction in Alphard is the form, comprised of parts for specification, representation, and implementation. The specifications provide at least the names of the functions of the form together with the types of their arguments and results. The representation defines the data structures used within the form to implement the abstraction. The implementation provides the specific mechanisms to realize the abstract functions. Only the specifications are visible to users of the form. In this way, Alphard emphasizes the functionality of the interface between modules, yet retains the capability to specify concrete realizations of the functions.

2.2 Abstraction In The Smite Computer Description Language

The primary module of abstraction to be used in SMITE is the DEVICE, which has structure, use, and concept derived from the Alphard 'form'. A DEVICE is composed of the following elements:

1. A header, which specifies the name, connections, and sizing parameters of the device.

2. A SPECIFICATIONS section, which describes the functionality of the device. This specification is presently given as a prose segment; current research is underway to replace the prose description with an axiomatic definition of the device function.

3. A REPRESENTATION section, which defines the static data base required to implement the device.

4. An IMPLEMENTATION section, which operationally defines the functions and operations of the device.

The DEVICE extension to SMITE provides a means for both creating new devices, and extending existing devices. Characteristics may be inherited from previous devices when an extended device is being defined.

Abstraction using devices provides a method of describing a "black-box" and its internal implementation. The user knows nothing of the device other than the functions it provides, the inputs it expects, and the outputs it produces. The implementation of the device knows nothing of its usage except the connection and sizing specifications provided to the device when it is instantiated.

To illustrate the use of the DEVICE abstraction the Mark 1 description is rewritten using abstractions to describe functional boxes within the computer. These boxes are then connected and the necessary control added. To provide the example the Mark 1 is artificially decomposed into the structure shown in Figure 1.

After defining this structure, we specified the function of each device used to build the complete computer. After functional specification, we then specified the implementation of the devices, including internal storage, devices, and data paths. One method available for specifying the function of a device is to add a new statement to the language. For example, a new statement is added by the specification of the PROGRAM-COUNTER device:

```
PROGRAM-COUNTER : DEVICE (Y:DEVICE, Z:DEVICE) <W> EXTENDS WORD
{ <- };
   SPECIFICATION
      'A PROGRAM-COUNTER provides internal storage for the
      program address register, and mechanisms to load and
      increment.'
      USES REGISTER;
      STATEMENT BUMP;
   REPRESENTATION
      DECLARE Q <1:W> REGISTER;
   IMPLEMENTATION
      BUMP: STATEMENT "BUMP A:ID"
         WHERE
            A: PROGRAM-COUNTER
```

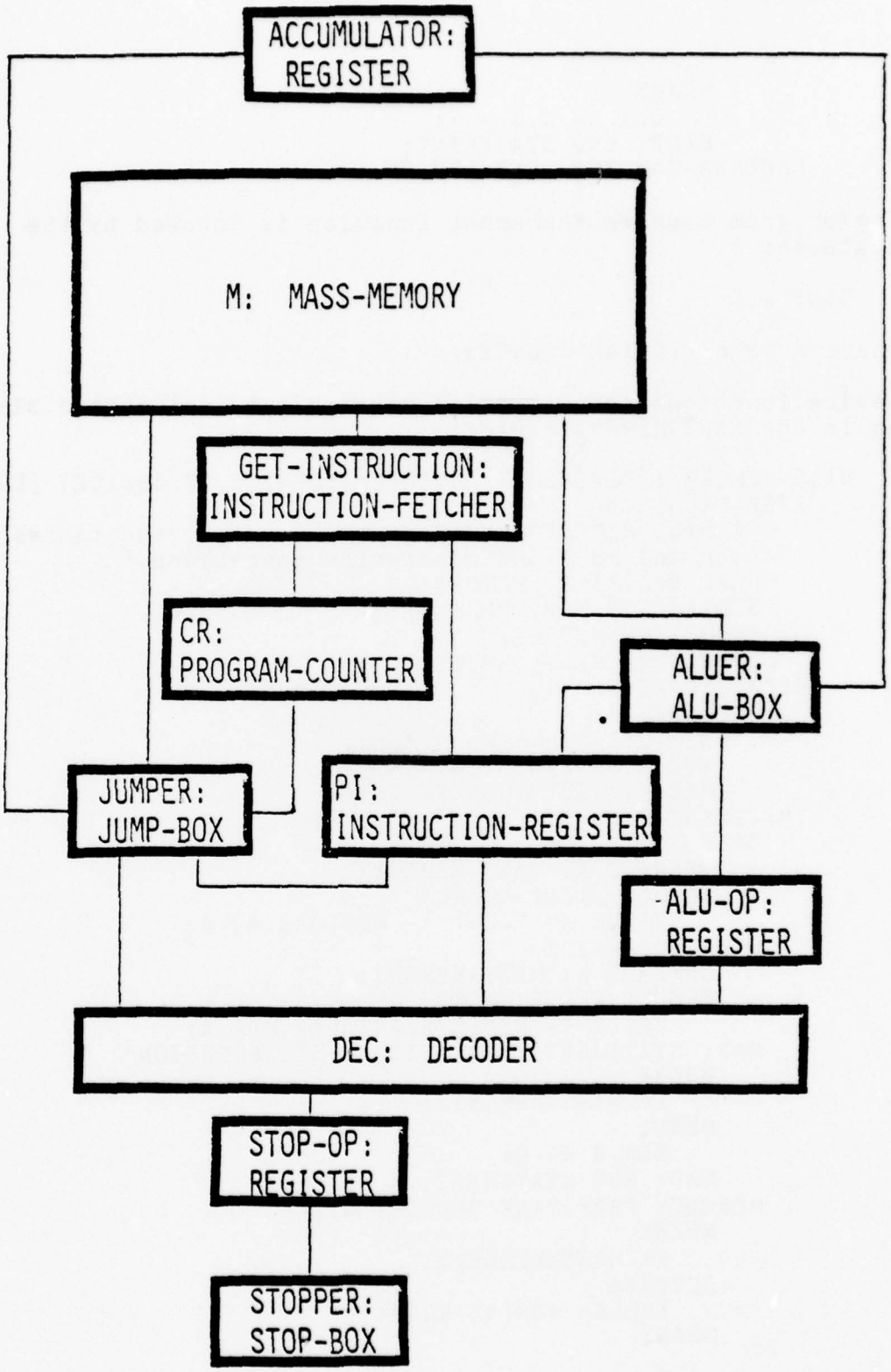


Figure 1
Mark 1

```
        MEANS
            Q.A <- Q.A + 1;
        BUMP: END STATEMENT;
    PROGRAM-COUNTER: END DEVICE;
```

The program counter increment function is invoked by the statement

```
BUMP A,
```

where A is a PROGRAM-COUNTER.

Device functions returning information are implemented similarly, as in the MASS-MEMORY device:

```
MASS-MEMORY : DEVICE (X:DEVICE, Y:DEVICE, Z:DEVICE) [L] <W>;
    SPECIFICATION
        'A MASS-MEMORY provides mass storage, associated ports
        (mar and mdr) and read/write operations.'
        USES REGISTER, MEMORY;
        STATEMENTS MAR, MDRIN;
        PRIMITIVE MDROUT;
        OPERATORS READ, WRITE;
    REPRESENTATION
        DECLARE
            MEM[0:L-1] <1:W> MEMORY,
            MDR<1:W> REGISTER,
            MAR <1:WIDTH(L)> REGISTER;
    IMPLEMENTATION
        OPERATOR READ
            OPERAND A: MASS-MEMORY;
            RESULT VALUE: WORD;
                VALUE <- MDR.A <- MEM[MAR.A].A;
        OPERATOR WRITE
            OPERAND A: MASS-MEMORY;
            RESULT VALUE: WORD;
                VALUE <- MEM[MAR.A].A <- MDR.A;
        MAR: STATEMENT "MAR A:ID <- B:EXPRESSION"
            WHERE
                A: MASS-MEMORY;
            MEANS
                MAR.A <- B;
            MAR: END STATEMENT;
        MDROUT: PRIMITIVE "MDROUT A:ID"
            WHERE
                A: MASS-MEMORY;
            RETURNS
                VALUE: WORD<1:W.A>;
            MEANS
```

```
            VALUE <- MDR.A;
         MDROUT: END PRIMITIVE;
      MDRIN: STATEMENT "MDRIN A:ID <- B:EXPRESSION"
         WHERE
            A: MASS-MEMORY;
         MEANS
            MDR.A <- B;
         MDRIN: END MACRO;
   MASS-MEMORY: END DEVICE;
```

The fetch of data from the memory data register after a read operation is defined as the primitive MDROUT, allowing the result of this function to be used as a part of a more complex operation.

New operators may also be defined for devices, such as in the INSTRUCTION-REGISTER device:

```
INSTRUCTION-REGISTER: DEVICE (V:DEVICE, X:DEVICE, Y:DEVICE,
Z:DEVICE) <W> EXTENDS WORD { <- };
   SPECIFICATION
      'An INSTRUCTION-REGISTER provides local storage for the
      current instruction, as well as the means for loading
      and fetching various sub-fields.'
      USES REGISTER;
      OPERATORS ADDRESS-FIELD, OP-CODE;
   REPRESENTATION
      DECLARE
         IR<1:W> REGISTER,
         OP-CODE <0:2> DEFINED IR <1:3>,
         ADDRESS <1:W-3> DEFINED IR<4:W>;
   IMPLEMENTATION
      OPERATOR     ADDRESS-FIELD
         OPERANDS A: INSTRUCTION-REGISTER;
         RESULT VALUE: WORD;
            VALUE <- ADDRESS.A;
      OPERATOR     OP-CODE
         OPERANDS A: INSTRUCTION-REGISTER;
         OPERATORS VALUE: WORD;
            VALUE <- OP-CODE.A;
   INSTRUCTION-REGISTER: END DEVICE;
```

The unary operator ADDRESS-FIELD accepts an INSTRUCTION-REGISTER type operand, and returns one of its sub-fields as the result.

In general, new operators are defined for unary and binary functions that return one value. New primitives are defined for functions that have more than two operands and return only one value. Statements are created for functions do not fall in the

previous two categories. The distinction between primitives (or operators) and statements is similar to the distinction between functions and subroutines in conventional programming languages. Statements are invoked only for their effect; primitives may have side effects, but also return a value.

A DEVICE abstraction describes the functions of the device. DEVICES are created by declaration of the device with all the necessary parameters supplied. More than one instance of a device may be declared. For example:

```
DECLARE
   PC0 (A,B) <5> PROGRAM-COUNTER,
   PC1 (A,B) <25> PROGRAM-COUNTER,
   PC2 (A,B) <16> PROGRAM-COUNTER,
   MEM0 (C,D,E) [1024] <60> MASS-MEMORY,
   MEM1 (C,D,E) [16384] <8> MASS-MEMORY;
```

One of the advantages of abstraction is that a device need only be specified once. Specific characteristics of the device may be left unspecified through the use of parameters until an instance of the device is declared. The use of parameters to permit the specification of device connections at declaration time is illustrated by the description of the DECODER device:

```
DECODER: DEVICE (JMP:JUMP-BOX, ALU:REGISTER, STOP:REGISTER,
IR:INSTRUCTION-REGISTER);
   SPECIFICATION
      'The instruction DECODER device decodes the primary Mark
      1 op-code into secondary op-codes suitable for execution
      by each of the functional boxes.  A DECODER expects to
      be connected directly into a jump box, op-code registers
      for alu box and stop box, and to an instruction
      register.'
      STATEMENT ENABLE;
   IMPLEMENTATION
      ENABLE: STATEMENT "ENABLE A:ID"
         WHERE
            A: DECODER;
         MEANS
            JMP-OPCODE <- 0;
            ALU <- 0;
            STOP <- 0;
            CASE OP-CODE IR;
               "JMP" JMP-OPCODE <- 1;
               "RJMP" JMP-OPCODE <- 2;
               "LDA" ALU <- 1;
               "STA" ALU <- 2;
               "SUB" ALU <- 3;
```

```
                "SUB" ALU <- 3;
                "SKP" JMP-OPCODE <- 3;
                "HLT" STOP <- 1;
                END CASE;
          DECODE:END STATEMENT;
    DECODER: END DEVICE;
```

The DECODER abstraction only specifies that the device is to be connected to devices of types JUMP-BOX and REGISTER, and not which devices. The declaration of the device serves to instantiate a DECODER and connect it to other devices.

Since the DEVICE abstraction only specifies the types of the connecting devices, the specification retains flexibility in the interconnection of devices. Frequently used devices need be defined and verified only once, and may be declared and connected in a system as required.

Following are the rest of the devices needed for a description of the MARK1.

```
STOP-BOX: DEVICE(OP:REGISTER);
   SPECIFICATION
      'This device halts the computer.  The device expects its
      op-code to be in an external register, and has two
      operations (halt and no-op).'
      USES EXTERNAL;
      STATEMENT ENABLE;
   IMPLEMENTATION
      ENABLE: STATEMENT "ENABLE A:ID";
         WHERE
            A: STOP-BOX;
         MEANS
            DECLARE
               STOP EXTERNAL;
            CASE OP.A;
               "NO-OP" NULL;
               "HALT" STOP;
               END CASE;
         STOP: END STATEMENT;
   STOP-BOX: END DEVICE;

ALU-BOX: DEVICE (OP:REGISTER, IR:INSTRUCTION-REGISTER,
ANYMEM:MASS-MEMORY, ACC:REGISTER) <W>;
   SPECIFICATION
      'This device is an alu functional box with the 4
      functions no-op, load accumulator, store accumulator,
      and subtract. The device expects its op-code to be in an
```

```
      external register.  Access to a memory and instruction
      register are needed, as is an external accumulator.'
      USES REGISTER;
      OPERATOR ENABLE;
   IMPLEMENTATION
      OPERATOR   ENABLE
         OPERANDS A:ALU-BOX;
         RESULT VALUE:WORD;
            CASE OP.A;
               "NO-OP" VALUE <- UNDEFINED;
               "LDA" VALUE <- ACC.A <- FETCH-OPERAND;
               "STA" STORE-OPERAND(VALUE <- ACC.A);
               "SUB" VALUE <- ACC.A <- ACC.A - FETCH-OPERAND;
      FETCH-OPERAND: PROCESSOR<1:W>;
         MAR ANYMEM <- ADDRESS-FIELD IR;
         READ ANYMEM;
         FETCH-OPERAND <- MDROUT ANYMEM;
         FETCH-OPERAND: END;
      STORE-OPERAND PROCESSOR(IN);
         DECLARE
            IN <1:W> REGISTER;
         MAR ANYMEM <- ADDRESS-FIELD IR;
         MDRIN ANYMEM <- IN;
         WRITE ANYMEM;
         STORE-OPERAND: END;
   ALU-BOX: END;

JUMP-BOX: DEVICE (PC:PROGRAM-COUNTER, IR:INSTRUCTION-REGISTER,
ANYMEM:MASS-MEMORY, ACC:REGISTER, Z:DEVICE);
   SPECIFICATION
      'This device provides a jump functional box with 4
      functions (no-op, jump, relative jump, and skip if
      negative).  The device expects to be connected to a
      program counter, instruction register, and mass memory.
      It has its own internal op-code register and provides
      the outside world a method of loading it.  This device
      also expects to be connected to an external register
      used as an accumulator.'
      USES REGISTER;
      STATEMENT ENABLE, JMP-OPCODE;
   REPRESENTATION
      DECLARE OP <0:1> REGISTER;
   IMPLEMENTATION
      ENABLE: STATEMENT "ENABLE A:ID"
         WHERE
            A: JUMP-BOX;
         MEANS
            CASE OP.A;
               "NO-OP" NULL;
```

```
                    "JMP" PC <- FETCH-JUMP;
                    "RJMP" PC <- PC + FETCH-JUMP;
                    "SKP" IF SE(ACC.A) < 0
                       THEN BUMP PC;
                       END IF;
                    END CASE;
              ENABLE: END STATEMENT;
           FETCH-JUMP: PROCESSOR<1:13>;
              DECLARE MEM-WORD <1:32> DATA,
                 ADDRESS-MASK <1:13> DEFINED MEM-WORD <20:32>;
              MAR ANYMEM <- ADDRESS-FIELD IR;
              READ ANYMEM;
              FETCH-JUMP <- MDROUT ANYMEM.ADDRESS-MASK;
              FETCH-JUMP: END;
           JMP-OPCODE: STATEMENT "JMP-OPCODE <- A:EXPRESSION"
              MEANS
                 OP <- A;
              JMP-OPCODE: END STATEMENT;
        JUMP-BOX: END DEVICE;

     INSTRUCTION-FETCHER: DEVICE (PC:PROGRAM-COUNTER,
     IR:INSTRUCTION-REGISTER, ANYMEM:MASS-MEMORY);
        SPECIFICATION
           'This device fetches an instruction from memory and
           places it in the instruction register.  It expects to be
           connected to a program counter, instruction register,
           and memory.'
           STATEMENT ENABLE;
        IMPLEMENTATION
           ENABLE: STATEMENT "ENABLE A:ID"
              WHERE
                 A: INSTRUCTION-FETCHER
              MEANS
                 DECLARE MEM-WORD <1:32> DATA,
                    INSTRUCTION-MASK <1:16> DEFINED MEM-WORD
                    <1:16>;
                 MAR ANYMEM <- PC;
                 READ ANYMEM;
                 IR <- MDROUT ANYMEM.INSTRUCTION-MASK;
              ENABLE: END STATEMENT;
        INSTRUCTION-FETCHER: END DEVICE;
```

Having specified all the component devices of the computer, we then instantiate and connect them together:

```
     DECLARE
        PI (JUMPER, GET-INSTRUCTION, DEC, ALUER) <16>
        INSTRUCTION-REGISTER,
        M (JUMPER, GET-INSTRUCTION, ALUER) [8192] <32> MASS-MEMORY,
```

```
      CR (GET-INSTRUCTION, JUMPER) <13> PROGRAM-COUNTER;
      ACCUMULATOR <0:31> REGISTER,
      GET-INSTRUCTION (CR, PI, M) INSTRUCTION-FETCHER,
      JUMPER (CR, PI, M, ACCUMULATOR, DEC) JUMP-BOX,
      ALU-OP<1:0> REGISTER,
      ALUER (ALU-OP, PI, M, ACCUMULATOR) <32> ALU-BOX,
      STOP-OP FLAG,
      STOPPER(STOP-OP) STOP-BOX,
      DEC(JUMPER, ALU-OP, STOP-OP, PI) DECODER;
```

The declarations specify the connections as parameters to the devices. For example, the declaration of JUMPER indicates that the device is connected to devices CR, PI, DEC, and M, and has access to the accumulator.

The complete computer is then described operationally using the interconnected set of devices:

```
   MARK1: PROCESSOR;
      USES
         INSTRUCTION-REGISTER, MASS-MEMORY, PROGRAM-COUNTER,
         INSTRUCTION-FETCHER, JUMP-BOX, REGISTER, ALU-BOX,
         FLAG, STOP-BOX, DECODER;
      DECLARE
         PI (JUMPER, GET-INSTRUCTION, DEC, ALUER) <16>
         INSTRUCTION-REGISTER,
         M (JUMPER, GET-INSTRUCTION, ALUER) [8192] <32>
         MASS-MEMORY,
         CR (GET-INSTRUCTION, JUMPER) <13> PROGRAM-COUNTER;
         ACCUMULATOR <0:31> REGISTER,
         GET-INSTRUCTION (CR, PI, M) INSTRUCTION-FETCHER,
         JUMPER (CR, PI, M, ACCUMULATOR, DEC) JUMP-BOX,
         ALU-OP<1:0> REGISTER,
         ALUER (ALU-OP, PI, M, ACCUMULATOR) <32> ALU-BOX,
         STOP-OP FLAG,
         STOPPER(STOP-OP) STOP-BOX,
         DEC(JUMPER, ALU-OP, STOP-OP, PI) DECODER;
      DO FOREVER;
         BEGIN;
            ENABLE GET-INSTRUCTION;
            ENABLE DEC;
            PARALLEL-BEGIN;
               ENABLE JUMPER;
               ENABLE ALUER;
               ENABLE STOPPER;
               PARALLEL-END;
            BUMP CR;
            END;
      MARK1: END;
```

The description of the Mark 1 itself contains only the top layer of abstraction which, coupled with the functionality of each device, provides a functional description of the computer. The description contains the inter-component connection scheme and the invocations for control functions.

The next lower level of abstraction provides the complete description of each device. These lower level descriptions contain the input/output specification and concrete implementation of each device. Another, lower level of abstraction could conceptually be added, in which the actual hardware items (chips, transistors, etc.) are used to build the primitive components of the computer.

3.0 Syntax and Semantics

This section contains the detailed syntax and semantics of the constructs necessary to implement the SMITE extensibility features discussed in this report and is organized as follows:

1. Overall extensibility description (Section 3.0)
2. DEVICE description (Section 3.1)
3. Syntax Macro description (Section 3.2)
4. Direct code description (Section 3.3)
5. Description of additional constructs needed to implement the extensibility features (Section 3.4)

The major vehicle of extensibility within SMITE will be the DEVICE. This mechanism is based on a simplified version of the Alphard "form" and consists of the following four sections:

1. Header
2. Specification
3. Representation
4. Implementation

The DEVICE capability is used to define a new SMITE data type. Two constructs are provided to support functions which operate upon the new type: operator and syntax macro. The operator construct can be used to define new binary and unary operations on the type. The compiler automatically provides the mechanism to extend the syntax it recognizes to implement this construct. If functions other than this capability supports are needed, syntax macros may be used. Syntax macros allow user definition of new compiler legal syntax which significantly increases the flexibility of the definition of functions to support the new types.

To efficiently implement areas of common code or provide modularity, helper processors may be used by either syntax macros or operators. Helper processors cannot be accessed outside of the DEVICE; however, they may be indirectly used through syntax macros. Helper processors are identical to basic SMITE processors with the following exceptions:

1. They can have the optional inline/closed specification. If neither is specified, the compiler decides how they are to be implemented.

2. Helper processors are considered to be within the scope of the DEVICE for context purposes, but cannot refer either to data in the representation section or parameters into the DEVICE. If any of these items are needed they must be passed as parameters into the helper processors.

3. Direct code blocks may appear within a helper processor.

4. The declaration part can contain a DEFAULT declaration.

5. Helper processors may not be nested.

Syntax macros, helper processors and operators are separate and autonomous entities within the implementation section. Operators or syntax macros defined within a DEVICE cannot be used by any operator, syntax macro, or helper processor within the same DEVICE. This is consistent with the notion of DEVICEs in that the outside world knows only the abstracted operators and syntax macros passed out, and the DEVICE knows only of its internal workings and parameters and operands passed in.

The expansion of operators and syntax macros at compile time is based on an optional inline/closed indicator. If the indicator does not appear, the complier makes a decision as how to expand the entity. No matter how they are actually expanded (inline or closed) they are treated like closed entities, and all parameters are passed either CBV or CBVR.

Operators and syntax macros are within the scope of the containing DEVICE and may refer to any data in the representation section and/or any formal parameters of the DEVICE. Since an operator or

syntax macro may have more than one instance of the DEVICE as parameters, there needs to be a method of qualifying representation section data items and formal parameters. The problem can be seen in the following example:

```
INTEGER:DEVICE<W>;
   . . .
   REPRESENTATION
      DECLARE I<1:W>REGISTER;
   IMPLEMENTATION
      OPERATOR +
         OPERANDS    A:INTEGER,B:INTEGER;
         RESULT      C:INTEGER;
   . . .
```

If the operator + is to add the two operands A and B together a method of specifying the concrete data representations of A and B is needed. This is because the implementation section can only manipulate the concrete data, and not the abstract data type INTEGER. The method of specification is to qualify the concrete data with a post-qualifier consisting of the operand name. For example the + operator can be coded as:

```
I.C <- I.A + I.B;
```

In the same manner, the value of the formal parameter W can be referenced within the operator or syntax macro definition as:

```
W.A   or     W.B
```

This allows reference to the value of the parameter for either instance of INTEGER.

This type of qualification is needed to refer to any concrete data representation of a DEVICE where an ambiguity arises as to which instantiation copy of the concrete data is desired. This ambiguity can occur where there is more than one parameter of the type of the DEVICE into either an operator or a syntax macro.

There will be a major division between the extensions provided by the DEVICE capability and the actual computer descriptions written in SMITE that use the extensions. DEVICEs will be maintained in a library to aid in this division and also to remove the need of repetitively coding the same extensions. Whenever a description is compiled the compiler needs to be notified where the library resides (host computer file). The description itself indicates which of the DEVICEs are needed from the library by way of the USES statement. The compiler must also provide the ability to create and maintain the device library.

The direct code capability is provided for by allowing its use within operators, syntax macros, and helper processors. Direct code is not allowed in the actual description itself.

3.1 DEVICE

3.1.1 Semantics

1. Header Section

A header is needed to identify the name of the device, an optional list of parameters for the device, and any inherited capabilities. The id specified as the device name is the name which is used as a type in declarations of instances of the device. This name must be specified in a USES clause before the new data (device) type (and any operators, primitives, and statements associated with it) may be used within a description. This id must not be a SMITE reserved word prior to declaration of the device, and becomes a SMITE reserved word within the scope of the device and any description containing a USES clause naming it.

The parameters identified in the device header are formal parameters, and may be used throughout the device description. They must be matched in type and quantity by each SMITE declaration using the device name as a data type.

The parameter list surrounded by parentheses is used to transmit a list of devices to which this device is connected. The elements of this list indicate the internal name and type of each device connected to the device being

described. If a legal connection can be made to several different entities one of the global types, DEVICE or WORD, may be used as the connection type indicator. The type WORD is used for a connection to any base SMITE type or a user defined device extending WORD, while the type DEVICE is used for a connection to any user defined device. The other two types of parameter lists have no pre-defined meanings, and may be used in any way. However, to conform to standard SMITE notation, the parameter list surrounded by wedges should be used to transmit values that have a width connotation, and the parameter list surrounded by square brackets should be used to transmit values that have a length connotation.

An extends clause specifies inherited capabilities of the new data type. In this way any new data type may build on a compatible existing data type and there is not a need for duplication of functions for the new type when the functions of the existing type will suffice. The source of the capabilities is the type specified in the clause. If no capabilities list appears, the new data type inherits all capabilities that exist for the type. If an inclusion capabilities list appears, only those capabilities listed are inherited. If an exclusion capabilities list appears, only the existing capabilities of the type not listed are inherited.

At least one data item of the type extended must be declared in the REPRESENTATION section of the new data type as a concrete data item. This is to insure that the inherited capabilities have a data item upon which they can operate. If more than one data item of the type extended exists there is a confusion as to which concrete data item the inherited capabilities are to operate on. This confusion is solved by preceding the extended type with an identifier specifying which of the concrete data items is to be used with inherited operators. For example:

```
WIDGET: <W> EXTENDS A:REGISTER {+, -}
   ...
   REPRESENTATION
      DECLARE A<1:W> REGISTER,
         COUNTER<0:7> REGISTER,
   ...
```

Whenever the inherited operators + or - are used with operands of WIDGET the concrete data item A will be the item operated upon.

A device trailer must be present to terminate the DEVICE and it must contain a label identical to the one in the corresponding device header.

2. Specification Section

A device description must contain a specification section which describes the functionality of the device, specifying everything needed to use the language extensions in the device, and identifying other devices employed in the description of the current device.

The specification contains a required text block that can be used as a prose functional description of the new data type, and/or the new operators, primitives, and statements. The prose should include complete descriptions of the external behavior of all language extensions in the device, and assumptions or conditions that must hold in using the extensions in SMITE code. The text block may be replaced with a formal functional specification of the DEVICE in the future, if the technology becomes available.

There must be a method of indicating which functions (syntax macros and operators) described within the DEVICE are used by the outside world. This is accomplished by employing a set of lists. Only functions that appear on these lists may be exported. The operators list identifies a set of tokens which are operators described in the DEVICE. The syntax macros list identifies a set of identifiers which are labels on syntax macros describing either new statements or new primitives in the DEVICE.

Another list is used to specify all data types needed within the scope of its definition. This list can appear in the specification section of a DEVICE and before the main processor in a SMITE program. By using the optional pseudo reference mechanism the user may give a data type an additional pseudo name. This pseudo name can then be used anywhere the data type is desired. For example:

```
...
USES FAST-REGISTER AS REG;
...
DECLARE A <0:15> REG;
```

would result in the declaration of data item A of type FAST-REGISTER.

3. Representation Section

A representation section is needed to define the static data base used in the implementation of the DEVICE. One copy of this data base is allocated for each SMITE declaration of the type described by the DEVICE. The data declared in the representation section may be referenced and manipulated in the descriptions of the new language extensions in the DEVICE. The data declared in this section are not directly accessible by SMITE code outside the device description (or by helper processors described within the device); this data may be indirectly accessed by using the language extensions described in the device.

4. Implementation Section

An implementation section is needed to contain the concrete implementations of all operators and Syntax Macros associated with a DEVICE. These implementations are realized through manipulation of operands and concrete data described within the representation section. This section also contains any helper processors necessary for these implementations.

An operator definition mechanism must be provided for the definition of new prefix unary and infix binary operators. These operators will take on the same precedence as the base SMITE unary and binary operators respectively. There must be a method of indicating whether the operator is implemented as in line or as closed and if this indication is missing it is to be left up to the compiler to decide. In all cases, the code associated with the operator is considered a logical entity and does not interact with any surrounding code or data items except through the operands and result mechanism. The operator name listed may redefine an existing operator or create a new one. This mechanism also indicates whether the operator is unary or binary by the presence of one or two operands respectively.

Besides indicating typing information for the operator implementation, the typing of operands allows the compiler to perform discrimination between multiple identical operators which perform different actions for different operand types. The type(s) of the operand(s) in the actual operator invocation must be compatible with the types specified in the operands clause. For example:

```
OPERATOR +
OPERANDS A:VECTOR,B:VECTOR
   and
OPERATOR +
```

```
OPERANDS A:DECIMAL,B:DECIMAL
```

There is no ambiguity if both +'s are available because the operand typing requirements need to be met before the implementation is actually invoked.

To further aid in implementation discrimination, optionally following the operands clause are any number of attribute requirements. These are compile time expressions which must all evaluate to boolean TRUE (in addition to the operand type checking mentioned above) before the implementation is invoked. These compile time expressions may perform functions such as placing restrictions on the width of either or both operands. If an ambiguity does result after all discrimination tests have been met the compiler reports an error and attempts recovery.

In addition to describing the attributes of the operands, a method of indicating the shape and attributes of the result of the operation must also be provided. The result clause accomplishes this function by listing a data item and any necessary attributes.

The operator clause must also contain the actual code for the implementation of the operator. The value of the operation is "returned" by storing into the data item named in the result clause somewhere within the actual code of the implementation.

3.1.2 Syntax

```
<device> ::=
   <device-header> <specification-section>
   [<representation-section>] <implementation-section>
   <device-trailer>

<device-header> ::=
   <device-label> DEVICE [<device-parameter-clause>]
   [<extends-clause>] ";"

<device-label> ::=
   <device-name> ":"

<device-name> ::=
   <id>

<device-parameter-clause> ::=
   (<type-square> [<type-point>] [<type-paren>]) /
   (<type-square> [<type-paren>] [<type-point>]) /
```

```
   (<type-point> [<type-square>] [<type-paren>]) /
   (<type-point> [<type-paren>] [<type-square>]) /
   (<type-paren> [<type-point>] [<type-square>]) /
   (<type-paren> [<type-square>] [<type-point>])

<type-square> ::=
   "[" <element-list> "]"

<element-list> ::=
   <element> ["," <element>]*

<element> ::=
   <id> [":" <id>]

<type-point> ::=
   "<" <element-list> ">"

<type-paren> ::=
   "(" <typed-id-list> ")"

<typed-id-list> ::=
   <typed-id> ["," <typed-id>]*

<typed-id> ::=
   <id> [<type-square>] [<type-point>] [<type-paren>] <type>

<type>::=
   ":" <type-name> [<capability-list>]

<type-name>::=
   <base-smite-type> / <device-name> / <pseudo-name> / WORD /
   DEVICE

<pseudo-name> ::=
   <id>

<capability-list>::=
   "{" ( <exclusion> / <inclusion> ) "}"

<exclusion> ::=
   ALLBUT <capability-string>

<capability-string> ::=
   <capability-name> [ "," <capability-name>]*

<capability-name>::=
   <operator-name>

<inclusion> ::=
```

```
   <capability-string>

<extends-clause> ::=
   EXTENDS <typed-id> [<capabilities-list>]

<specification-section> ::=
   SPECIFICATION <text-block>
   [<operators-list> /
      <statement-macros-list> /
      <primitive-macros-list> /
      <uses-list> ]*

<text-block> ::=
   "'" <any-legal-character-except '> "'"

<operators-list> ::=
   (OPERATOR / OPERATORS ) [ IS / ARE ] <operator-name> ["," 
   <operator-name]* ";"

<operator-name> ::=
   <old-operator> / <new-token>

<old-operator> ::=
   <binaryop> / <unaryop>

<new-token> ::=
   any string consisting of legal token characters that is not
   an old token

<statement-macros-list> ::=
   ( STATEMENT / STATEMENTS ) [ IS / ARE ] <macro-name> ["," 
   <macro-name>]* ";"

<macro-name> ::=
   <id>

<primitive-macros-list> ::=
   ( PRIMITIVE / PRIMITIVES ) [ IS / ARE ] <macro-name> ["," 
   <macro-name>]* ";"

<uses-list> ::=
   USES <uses-element> ["," <uses-element>]* ";"

<uses-element> ::=
   <base-smite-type> /
   <device-name> /
   <pseudo-reference>

<pseudo-reference> ::=
```

```
   <pseudo-name> AS (<device-name> / <base-smite-type>)

<representation-section> ::=
   REPRESENTATION [<extended-decdriver>] *

<extended-decdriver> ::=
   [ DECLARE <extended-decphrase>
      [ "," <extended-decphrase>]* ";" ]*

<extended-decphrase> ::=
   <decphrase> / <temporary-declaration>

<implementation-section> ::=
   IMPLEMENTATION [<op-def-statement>]* [<operator-definition>
   / <syntax-macro-definition> / <helper-processor>]*

<operator-definition> ::=
   [ INLINE / CLOSED ] OPERATOR <operator-name>
   <operands-clause> <result-clause> <operator-clause>

<operands-clause> ::=
   OPERANDS <typed-id> ["," <typed-id>]
   [<attribute-requirement>]* ";"

<result-clause> ::=
   RESULT <typed-id>

<operator-clause> ::=
   <extended-decdriver> <operator-body>

<operator-body> ::=
   [<statement> / <direct-code-block>]+

<helper-processor> ::=
   [ INLINE / CLOSED ] <procheader> <extended-decdriver>
   <helper-processor-body> <helper-processer-trailer>

<helper-processor-body> ::=
   [<statement> / <direct-code-block>]+

<helper-processor-trailer> ::=
   [<label>] END ";"

<device-trailer> ::=
   <device-label> ( ENDDEVICE / END DEVICE ) ";"
```

3.2 Syntax Macro

3.2.1 Semantics

A syntax macro is composed of five parts:

1. HEADER clause
2. TEMPLATE clause
3. WHERE clause
4. MEANS clause
5. END clause

3.2.1.1 HEADER clause

The HEADER clause contains the label. It must also appear on the END clause and be identical.

3.2.1.2 TEMPLATE clause

The TEMPLATE clause specifies the syntax of the macro call. It contains stand alone identifiers and qualified identifiers. The qualified identifiers are formal parameters into the syntax macro and will be replaced by the actual parameters when calls to the syntax macro are compiled. The parameters to a syntax macro can be one of two syntactic entities: ID or EXPRESSION. The formal parameters are used throughout the rest of the syntax macro wherever information or action concerning the parameters is needed.

The stand alone identifiers are either already SMITE keywords or they are added to the keywords list. The keywords in the template clause must be such that the macro call is unambiguous, when added to the rest of the syntax.

3.2.1.3 WHERE clause

The WHERE clause allows the statement of semantic requirements levied against the syntax macro parameters. These requirements must be met for the macro to be expanded.

The semantic requirements can be broken into two areas: type requirements and attribute requirements.

Syntax macro parameter type requirements are stated in a type requirement statement by listing the

parameter in question followed by the type and an optional capabilities list. The capabilities list can explicitly require certain capabilities to either be present for the actual parameter or strip away certain capabilities of the actual parameter within the context of the syntax macro. Using the capabilities list the user can verify that the parameters into the syntax macro support the operations needed within the syntax macro. They can also strip away properties, such as transfer, so as to retain the integrity of the parameter. Using this type requirement mechanism various syntax macros consisting of the same keywords can exist and the type requirements of the parameters can be used to decide which macro to actually expand for any specific call.

In addition to the capabilities list, the type requirement statement also allows the user to label the various attributes a data type can have: square bracket, angle bracket and parenthesized attributes. This attributes list is optional. If present, it must correspond to the attributes of the corresponding actual parameter or a subset of them. This attribute labeling mechanism allows the syntax macro to refer to the attributes of data types that are not being created by the specific form the syntax macro is within.

The attribute requirements consist basically of compile time expressions. Each requirement is an expression that is evaluated and all must evaluate to a true value for the syntax macro to be expanded. These compile time expressions are composed of the normal expression entities (such as binary operators, unary operators, constants).

In addition, the functions WIDTH, LENGTH, MAX and MIN can be used wherever a constant is allowed. The parameters of MAX and MIN must also be compile time expressions while the parameters to WIDTH and LENGTH can be identifiers. They are also used to "talk about" the various attributes of the syntax macro parameters labeled as described above. The identifiers may need qualification to uniquely define the data required. A syntax macro parameter of type EXPRESSION cannot be used as a parameter to LENGTH. (Note: A data item that was

not defined with a length attribute has a length of 0).

3.2.1.4 MEANS clause

The MEANS clause actually describes the "meaning" or action of the syntax macro, It is basically a SMITE processor minus the PROCESSOR statement and declarations for the parameters. It can have a declaration part and the declarations are identical to standard SMITE declarations.

The MEANS clause can contain any standard SMITE statement. In addition, any syntax macro formal parameter typed ID can appear anywhere an identifier is legal and any syntax macro formal parameter typed EXPRESSION can appear in any place an expression is legal.

When macro expansion occurs, as permitted by the TEMPLATE and WHERE clauses, the formal parameters within the macro body (MEANS clause) are replaced with the actual parameters and that instance of the macro is then ready for the rest of the compilation process.

3.2.2 Syntax

```
<syntax-macro-definition> ::=
   <macro-header-clause> <template-clause> [<where-clause>]
   [<returns-clause>] <means-clause> <end-clause>

<macro-header-clause> ::=
   <macro-label> [ INLINE / CLOSED ] ( STATEMENT / PRIMITIVE )

<macro-label> ::=
   <macro-name> ":"

<template-clause> ::=
   "'" [<template-element>]+ "'"

<template-element> ::=
   <token> / <sm-parameter>

<token> ::=
   <old-token> / <new-token>

<old-token> ::=
   <old-operator> / <reserved-word>

<reserved-word> ::=
   <old-reserved-word> /  <new-reserved-word>
```

```
<old-reserved-word> ::=
   AND / BEGIN / CASE / CLOCK / DATA / DEBUG / DECLARE /
   DEFAULT / DEFINED / DO / DOWN / ELSE / END / ENDCASE /
   ENDIF / ESCAPE / EXTERNAL / FLAG / FOR / FOREVER / IF / IN
   / LIGHT / MEMORY / MICROSECONDS / MILLISECONDS / MS /
   NANOSECONDS / NOT / NS / NULL / OR / PARALLEL-BEGIN /
   PARALLEL-END / PORT / PROCESSOR / REGISTER / S / SE /
   SECONDS / SLA / SLC / SLL / SRA / SRC / SRL / STEP / SWITCH
   / THEN / TO / UNTIL / UP / US / WHILE

<new-reserved-word> ::=
   <new-device-reserved-word> / <new-sm-reserved-word>

<new-device-reserved-word> ::=
   ALLBUT / ARE / AS / DEVICE / DEVICES / ENDDEVICE / EXTENDS
   / IMPLEMENTATION / IS / OPERANDS / OPERATOR / OPERATORS /
   REPRESENTATION / RESULT / SPECIFICATION / USES / WITHIN /
   WORD

<new-sm-reserved-word> ::=
   CLOSED / COPY / DECODE / DIRECT / ENDPRIMITIVE /
   ENDSTATEMENT / EXPRESSION / ID / INLINE / LENGTH / MAX /
   MEANS / MIN / OPDEF / PRIMITIVE / RETURNS / SMITE /
   STATEMENT / TEMPORARY / WHERE / WIDTH

<sm-parameter> ::=
   <id> ":" <typer>

<typer> ::=
   ID / EXPRESSION

<where-clause> ::=
   WHERE <where-list> ";"

<where-list> ::=
   <where-statement> ["," <where-statement>]*

<where-statement> ::=
   <semantic-requirements>

<semantic-requirements> ::=
   <typed-id> / <attribute-requirement>

<attribute-requirement> ::=
   <compile-time-expression>

<returns-clause> ::=
   RETURNS <typed-id> ";"
```

```
<means-clause> ::=
   MEANS <extended-decdriver> <means-body>

<means-body> ::=
   [<sm-statement>]+

<sm-statement> ::=
   <statement> / <direct-code-block>

<end-clause> ::=
   <macro-label> ( END ( STATEMENT / PRIMITIVE ) /
   ENDSTATEMENT / ENDPRIMITIVE ) ";"
```

3.3 Direct Code

3.3.1 Semantics

A method is needed to declare a SMITE variable used to communicate between SMITE code and direct code. This is achieved through the use of the TEMPORARY data type. A TEMPORARY declaration designates that the declared identifier is to be maintained in 1, 2, or 4 QM registers (depending on declared width) throughout the scope of its declaration The expression indicating the width (in bits) must evaluate to one of the following numbers: 18, 36, 72. The optional ability to declare which specific QM registers to actually use is provided. If the register designation does not appear the compiler selects an available local store register(s). The expression used to indicate the specific QM register(s) must evaluate to a number between 0 and 31, inclusive.

Although identifiers of type TEMPORARY may be used like other declared identifiers in SMITE statements, they are the only SMITE identifiers that may be referenced in direct code, where they may appear as register designators in microcode instructions. It will appear to the user that the QM register(s) associated with a TEMPORARY identifier is not used in the generation of compiled code within the scope of the TEMPORARY declaration, other than for references to the TEMPORARY identifier. However the compiler has the option of using temporary storage if the register(s) is needed for other purposes.

A method is needed to notify the compiler of new microinstructions to be used in direct code. This is accomplished by using an OPDEF statement which informs the

SMITE compiler of the format of new microinstructions that are to be used in direct code and that have been separately entered into the QM system.

For each new microinstruction the following information is needed:

1. The instruction mnemonic which is used within the direct code.

2. The instruction op code which indicates where the instruction resides within the MULTI instruction set.

3. The instruction format type which indicates to the compiler how to assemble the instruction by indicating the type and number of operands.

Alternatively, the information needed to define a new microinstruction may be obtained from a text file maintained in the host file system. This file is named by the system file name of a copy clause, and is expected to contain the image of any number of op def statements.

A direct code block is assumed to be a reducible single-entry, single-exit "node". Labels declared in microinstruction statements are known only in the containing direct code block. SMITE identifiers of type TEMPORARY are the only SMITE identifiers that may be referenced in microinstructions, and they may appear anywhere a register is valid in an instruction. Also, a compile time constant may be used in a direct code block anywhere a constant is valid as an operand in a microinstruction.

Because of the difficulty of determining the effect of direct code references to absolute and relative addresses, the SMITE compiler maintains no set/use information over a direct code block, except for SMITE statements interspersed within the direct code block.

A microinstruction is a member of the subset of MULTI microcode instructions, as augmented by the new microcode instructions introduced by op def statements known within the scope of the direct code block. TEMPORARY references may be involved in compile time expressions for reference to TEMPORARY declarations needing more than one register, in which case the TEMPORARY reference in direct code designates the first assigned register. For example:

```
DEC  A  TEMPORARY<36>;
```

```
    . . .
    LDMSX A,...    ; references the first register allocated to
    A
    LDMS A+1,...   ; references the second register.
```

A method of interspersing SMITE statements within a direct code block is provided. Direct code labels are known around the interspersed SMITE statements in one direct code block. Across these SMITE statements, the compiler guarantees preservation of only those QM registers associated with TEMPORARY identifiers.

As an alternative decoding method to the CASE statement the DECODE statement is provided. The primary differences are that the width of the selector expression does not determine the number of selected statements, and that the user determines the method of decoding by providing the decode algorithm in nanocode which is separately loaded into the QM system.

When the compiler detects a DECODE statement, the following actions occur:

1. generate code to evaluate the expression (selector);

2. verify that the number of statements between DECODE and END DECODE (selected statements) is equal to the decode length (if present);

3. compile all selected statements with a transfer to a common collecting point (statement after END DECODE) at the end of each;

4. generate an address table with an entry that points to the beginning of the compiled code for each selected statement;

5. encode the MULTI instruction DECODE and indicate the location of the value of the selector and the beginning of the statement table by operands to the instruction.

It is up to the user to actually write the nanocode for MULTI instruction DECODE to decode the selector and transfer to one of the selected statements indicated in the statement table. This nanocode could also provide additional actions such as instruction fetch for the described machine.

The following actions occur when the DECODE statement is executed:

1. evaluate selector;

2. execute the MULTI instruction DECODE.

The actual op code value of the DECODE instruction is TBD.

The DECODE statement generates a context block and may be labeled. If labeled, both labels of the decode-statement syntax must appear and be identical.

3.3.2 Syntax

```
<temporary-declaration> ::=
   <id> TEMPORARY <temporary-width>
   [<temporary-defined-phrase>]

<temporary-width> ::=
   "<" <compile-time-expression> ">"

<temporary-defined-phrase> ::=
   DEFINED <qm-register-designator>

<qm-register-designator> ::=
   <compile-time-expression>

<op-def-statement> ::=
   OPDEF <op-def-clause> ["," <op-def-clause>]* ";"

<op-def-clause> ::=
   <new-multi-instruction-clause>
   / <copy-clause>

<new-multi-instruction-clause> ::=
   <instruction-mnemonic> <instruction-op-code>
   <instruction-format-type>

<instruction-mnemonic> ::=
   <id>

<instruction-op-code> ::=
   <compile-time-expression>

<instruction-format-type> ::=
   a SMITE assembler instruction type

<copy-clause> ::=
   COPY <system-file-name>

<system-file-name> ::=
```

```
   a host operating system file name

<direct-code-block> ::=
   DIRECT ";" <direct-statement-list> SMITE ";"

<direct-statement-list> ::=
   [<direct-statement>]+

<direct-statement> ::=
   <micro-instruction> / <direct-smite-statement>

<micro-instruction> ::=
   [<label>] <micro-operator> <micro-operand-list>
   <micro-comment> ";"

<micro-operator> ::=
   <id>

<micro-operand-list> ::=
   <micro-operand> [ "," <micro-operand>]*

<micro-operand> ::=
   <id> / <compile-time-expression>

<micro-comment> ::=
   [<token>]*

<direct-smite-statement> ::=
   "#" <statement>

<decode-statement> ::=
   [<decode-length>] <expression> ";"
   [<statement> / <nullstatement>]*
   [<label>] (END DECODE/ENDDECODE) ";"

<decode-length> ::=
   "[" <number> "]"
```

3.4 Additional Changes

Following are the changes to SMITE necessary to implement the extension features described in previous paragraphs:

1. The SMITE program structure needs to be redefined. Any number of uses lists may appear before the main processor to

indicate which DEVICES from the DEVICE library the description will use.

```
<smite-program> ::=
   [<uses-list>]* <processor>
```

2. The addition of compile time expressions. Compile time expressions are identical to SMITE expressions, with the exception that they are evaluated at compile time using 2's complement arithmetic and can be used wherever a constant is allowed. LENGTH, WIDTH, MIN, and MAX are compile time functions that are a part of the compile time expression mechanism. The result of the MAX(MIN) function is the maximum(minimum) value of the parameters, which must be compile time expressions. The result of the LENGTH(WIDTH) function is the length(width) attribute of the parameter which must be an identifier declared as a data item. A data item that was not defined with a length attribute has an implied length of zero and can be used as a parameter to LENGTH. Additionally an expression, which has an implied width, may be the parameter to WIDTH.

```
<constant> ::=
   <compile-time-expression>

<compile-time-expression> ::=
   <compile-time-term> [<binaryop> <compile-time-expression>]

<compile-time-term> ::=
   [<unaryop>] <compile-time-factor>

<compile-time-factor> ::=
   <compile-time-primitive> [ "//" <compile-time-factor>]

<compile-time-primitive> ::=
   <compile-time-constant> /
   "(" <compile-time-expression> ")"

<compile-time-constant> ::=
   <width> / <min> / <max> / <number> / <length>

<width> ::=
   WIDTH "(" <qual-id> ")"

<length> ::=
   LENGTH "(" <qual-id> ")"

<min> ::=
   MIN "(" <param-list> ")"
```

```
<max> ::=
   MAX "(" <param-list> ")"

<param-list> ::=
   <compile-time-expression> ["," <compile-time-expression>]*
```

3. A SMITE statement can now be any statement plus any new user defined statement. The statement macro reference cannot have a label; i.e., a new statement cannot define a context block.

```
<statement> ::=
   [<label>] <contextstatement> /
   <notcontextstatement>  ";" /
   <statement-macro-reference> ";"

<statement-macro-reference> ::=
   any reference to a new statement defined by using the
   Syntax Macro mechanism
```

4. A SMITE primitive can now be any primitive, any new user defined primitive, or a compile-time-expression.

```
<primitive> ::=
   <compile-time-expression> /
   <idprimitive> /
      "(" <expression> ")" [<extract>] /
   <primitive-macro-reference>

<primitive-macro-reference> ::=
   any reference to a new primitive defined by using the
   Syntax Macro mechanism
```

5. The DEVICE extension adds three new attributes to a data type which are parameters into the DEVICE. These parameters are used to provide sizing and connection information to the device. To allow declarations to specify the attributes or parameters, the following changes are made to the syntactic entity decphrase. The syntax for declaring new data types is slightly different than that for declaring base SMITE data types. The length-declaration is still present to allow the user to declare arrays of new data items but following that are two of the optional parameter types into a DEVICE, paren-param and point-param (in either order). These two optional parameter types are followed by the new data type name (new-type) and the optional square-param. The square-param follows the data type name to avoid confusion with the length-declaration, which was left in its original location to provide compatibility with descriptions written in Version 1 SMITE.

```
<decphrase> ::=
   <base-type-declr> / <new-type-declr>

<base-type-declr> ::=
   <id> [<length-declaration>] [<width-declaration>]
   [<base-smite-type> / <pseudo-name>] [<defined-phrase>]

<new-type-declr> ::=
   <id> [<length-declaration>] [<new-type-parameters>]
   [<new-type>] [<square-param>]

<new-type-parameters> ::=
   <point-param> [<paren-param>] /
   <paren-param> [<point-param>]

<point-param> ::=
   "<" compile-time-expression-list> ">"

<compile-time-expression-list> ::=
   <compile-time-expression> [ "," <compile-time-expression>]*

<paren-param> ::=
   "(" <id-list> ")"

<id-list> ::=
   <id> [ "," <id>]*

<square-param> ::=
   "[" <compile-time-expression-list> "]"

<new-type> ::=
   <device-name> / <pseudo-name>
```

6. This redefinition of id-primitive is only active within the IMPLEMENTATION section of a DEVICE.

```
<id-primitive> ::=
   <qual-id> [<reference>]
```

7. A contextstatement may now also be a DECODE statement.

```
<contextstatement> ::=
   BEGIN <serialcontext> /
   IF <ifstmnt> /
   DO <dostmnt> /
   CASE <casestmnt> /
   IN <instmnt> /
   PARALLEL-BEGIN <parallelcontext> /
   DECODE <decode-statement>
```

4.0 Implementation Implications

Parsing syntax macro definitions and their use presents a problem for predictive parsers - figuring out when to backup could be hard. The alternative is a reductive parser. An answer might be a bottom-up reductive parser, which would not be confused by these constructs. The technology exists on which to base the Advanced SMITE extensible parser [11] if this approach is taken.

Another possible solution for the parse problem might be a top-down predictive parser that "carries along" all possible parses of an entity in parallel while doing the syntactic analysis. In this way back up need never be performed. When a correct parse path is discovered the rest of the paths are simply "thrown away". One consequence of this approach is that the syntactic and semantic processing within the parser must be completely separate (perhaps only on a statement by statement basis).

Further discussion about the parser for extensibility showed that what is needed is context-dependency-resolvable syntactic ambiguity, not the ability to handle truly ambiguous grammars. E.g., parsing

IN <exp> + <exp>

as a new syntactic entity is not required.

An ambiguity may be created if a syntax macro is defined using words that already are reserved and no new keywords. Extensions that cause the language to be ambiguous may be tolerable because the ambiguous constructs may never be encountered in actual use. Since there is no way of pre-determining if syntax macro extensions cause the "new" language to be ambiguous, the best that can be done is to find the ambiguity in a specific instance, report it and reject the entity.

There is a general problem with processing aggregate data types. Since there can be arrays of DEVICEs there is a possibility that

array specifications may be used recursively (the concrete data representation may also contain an array). If this is not handled properly severe object code inefficiency may result.

5.0 SMITE Application Support Software Requirements

The SMITE Application Support System (SASS) is in many respects a primitive system. From our experience, it is adequate for the development, testing, and use of SMITE emulations. However, it requires a dedicated user to obtain necessary data from the emulator and perform appropriate conversions and data mappings. Many chores must be performed manually by the user which could be automated into SASS. The following paragraphs describe limitations to SASS and our proposed solution.

5.1 Statement Level Stepping

During emulator checkout, it is desirable to breakpoint the emulator at selected SMITE statements and to step through the SMITE program one statement at a time. The user currently has to add calls to the external STEP routine to force predetermined breakpoints at SMITE statements. Any additional program steps must be inserted by examining the microcode produced by the compiler, and placing a microbreakpoint at the appropriate location. This is a tedious and error-prone activity, which presumes the user is familiar with the QM-1 instruction set and the SMITE code generation sequences.

This process can be automated by allowing SASS to insert microbreakpoints from a table generated by the compiler which maps SMITE statements to emulation addresses. Program stepping can be performed by using the same table and verify that no QM-1 branch instructions occurred for the current SMITE statement.

5.2 Symbol Display and Modify

SMITE variables may be stored in main memory or control store and may occur as subfields of a QM-1 pair of words. To modify or display SMITE variables, the user must obtain the QM-1 address and bit positions of the variable from the allocation map. For modification operations, the user must clear the subfield reserved for the variable, convert the input data, and insert it into the subfield. For display operations, the user must extract the subfield and convert it into the appropriate display format.

This process could be automated in SASS, using the symbol table

generated by the compiler. For data modification, the input data would contain a format indication (e.g.X'FF') and SASS would convert and store the data. For display operations, an input would be created which added the variable to the state display and specified its display format. The symbol table data would then be used to include the converted symbolic data in the state display.

5.3 Traps Based on Data Storage

A very valuable debug feature is the capability of trapping when a particular variable is stored into. This is extremely useful in determining the conditions which case erroneous emulator action to occur or errors in the emulator itself.

To accomplish this the compiler must produce a table of addresses where each variable is changed. From this table, it is easy to cause the emulator to return control to SASS when a variable is changed or even when the value falls in a specified range.

5.4 SASS Library

SASS contains no generalized I/O interface. Each emulated I/O device must be simulated by a new SMITE external and the existing I/O simulations are oriented to one specific emulation.

SASS forces the computer description to list all external statements in a specific sequence or else rewrite the SASS handler for emulator recall conditions. A desirable solution is for SASS to call a routine written in SIMPL-Q existing on a library which has the same name as the external (i.e. the SMITE external OPSTEP would be processed by a SASS routine OPSTEP).

5.5 User Nanocode

If the emulation requires nanocode for new microinstructions used in direct code or in the DECODE statement, the user must insert the nanocode on the system cartridge. This obviously limits the flexibility of the system since all users must coordinate nanocode additions. The preferred solution is for the user to

specify his nanocode file on disc and have SASS load it into nanostore, verifying that it is loaded in free nanostore locations.

6.0 Conclusions and Applications

Abstraction has proven to be a powerful concept for modularizing software, improving the characteristics of flexibility, constructibility, maintainability, and reliability in software systems. The DEVICE abstraction capability in SMITE provides the same abilities within the hardware description language domain. DEVICEs allow the separation of the concepts of functionality from those of implementation within a description. The improved clarity and precision of description resulting from this use of abstraction technology in hardware descriptions has value in many areas. Two examples of the application of the DEVICE abstraction follow:

6.1 Hardware Security

Computer security describes the theory and practice of restricting access to information to properly authorized persons or systems. Work in the field has been concerned with software and procedures for accomplishing that mission. Secure operation of the hardware is tacitly assumed. For the same reasons that correctly specified software requires verification to demonstrate correct implementation, so does correctly specified hardware require a similar verification to demonstrate its correctness.

Hardware verification is extremely difficult or impossible with commonly used technology. Correlation of a prose machine architecture description to boolean equations or to wire lists is a mammoth effort, and provides little assurance that no hidden effects can compromise security.

A computer description language, when combined with abstraction concepts, provides a vehicle for partitioning the verification effort into manageable units, and for demonstrating equivalence between the hardware and its specification. A methodology for hardware verification could be based on the following steps, and could be performed during hardware design and development.

1. Partition the computer into devices. Specify the function of each device, and describe the register transfer implementation of the complete computer using these devices as components.

2. Verify the register transfer implementation of the computer against the formal functional specification assuming correct implementation of all devices.

3. Specify the register transfer implementation of each device used to construct the computer, and verify each implementation against its corresponding specification. Further device partitionings may be required; the implementation and verification process is repeated until all devices are completely specified, implemented, and verified.

4. Each register transfer description must now be realized in hardware. The hardware implementation is devised for each device, and is verified for equivalence to the register transfer implementation specification.

Once this process is complete, the hardware is known to be equivalent to the overall computer formal specification. Software and procedures may now be proven to operate correctly on the hardware using the methods developed by Crocker [12].

6.2 Automatic Implementation of Compiler Code Generators

Abstract specification is also useful in the automatic development of compiler code generators. Work by Newcomer [13] and Fraser [14] has shown that automated code generator production is feasible given an understanding of the machine operation. Automation of this understanding, which generally consists of deriving a set of input/output relations for each machine instruction, is difficult or impossible when the relations must be derived from conventional computer descriptions. The following example illustrates the point.

```
DECLARE
   INTER-CARRY FLAG,
   C FLAG,
   A <7:0> REGISTER,
   B <7:0> REGISTER,
   RESULT <7:0> REGISTER;
INTER-CARRY//RESULT<3:0> <- 0//A<3:0> + B<3:0>;
C//RESULT<7:4> <- INTER-CARRY + 0//A<7:4> + B<7:4>;
IF (RESULT<3:0> > 9) OR INTER-CARRY
   THEN RESULT <- RESULT + 6;
   ENDIF;
IF (RESULT<7:4> > 9) OR C
   THEN RESULT <- RESULT + X'60';
   ENDIF;
```

The example is a specific implementation of a two digit decimal addition without carry. An automatic code generator would be required to detect the particular algorithm as being decimal addition in order to know the function of the

instruction, even though that information was known to the hardware designer. A description using abstraction, however, allows the designer to express this added information.

```
DECIMAL : DEVICE <W> EXTENDS WORD { <-,};
   SPECIFICATION
      'This is a decimal type extension that provides for
      the declaration of unsigned decimals w decimal digits
      wide. It also provides add and subtract operations
      without carry on decimal types of the same width.'
      OPERATORS  + , - ;
      USES  REGISTER, FLAG;
   REPRESENTATION
      DECLARE  J <4*L:1> REGISTER;
   IMPLEMENTATION
      OPERATOR  +
         OPERANDS  A,B:DECIMAL;
         RESULT  VALUE:<WIDTH(A)*4:1> WORD;
            DECLARE C FLAG, MASK <4:1> DATA;
            VALUE <- 0;
            C <- 0;
            DO FOR 1 TO WIDTH(A);
               BEGIN
                  VALUE <- SLL(VALUE,4) OR
                  ADD(J.A.MASK,J.B.MASK,C);
                  J.A <- SRC(J.A,4);
                  J.B <- SRC(J.B,4);
                  END;
      OPERATOR   -
         OPERANDS  A,B:DECIMAL;
         RESULT  VALUE:<WIDTH(A)*4:1> WORD;
            DECLARE C FLAG, MASK<4:1> DATA, TEMP <4:1>
            REGISTER;
            C <- 0;
            VALUE <- 0;
            DO FOR 1 TO WIDTH(A);
               BEGIN;
                  IF J.B = 0
                     THEN TEMP <- 0;
                     ELSE TEMP <- B'1010' - J.B.MASK;
                     ENDIF;
                  VALUE <- SLL(RESULT,4) OR ADD(J.A.MASK,
                  TEMP, C);
                  J.A <- SRC(J.A,4);
                  J.B <- SRC(J.B,4);
                  END:
      ADD: PROCESSOR <3:0> (IN0,IN1,CARRY);
         DECLARE IN0 <3:0> REGISTER, IN1 <3:0> REGISTER,
         CARRY FLAG;
```

```
            CARRY//ADD <- 0//IN0 + IN1 + CARRY;
            IF CARRY//ADD > 9
               CARRY//ADD <- CARRY//ADD + 6;
               ENDIF;
            ADD:END;
      DECIMAL: END DEVICE;
```

With the abstract device, decimal addition is described as follows:

```
   DECLARE
      A<2> DECIMAL,
      B<2> DECIMAL,
      RESULT<2> DECIMAL;
   RESULT <- A + B;
```

Using this description, the functionality known to the designer is specified, and (once a formally recognizable specification language is invented) may be used directly by an automatic code generator. The development of automated code generator producers is greatly simplified by elimination of the need to recreate that knowledge.

In these two applications, and in others to which abstraction of hardware specification applies, it is not necessary to create a complete new device set for each description. Instead, in the same way that subroutine libraries are developed and maintained in the software domain, device catalogs can be created in the hardware domain. Such catalogs might contain registers and standard ALUs as basic elements. More complex elements could include associative memories, floating point units, CPUs, and complete processors.

APPENDIX A

Working Notes

1.0 Analysis of Alphard Technology

This section contains the working notes compiled during the study of the Alphard 'form' and its application to SMITE. This study was concerned with the following tasks:

1. Understand the Alphard language and underlying concepts in full detail.

2. Summarize fundamental concepts and ideas of Alphard, including analysis of applicability to SMITE.

1.1 Analysis of Extensibility Incorporating Alphard Technology

Preliminary investigations into the question of designing the syntax and semantics of the SMITE language reveal that a strong initial approach is to utilize syntax macro and direct microcode techniques, with related statements grouped into abstractions using something akin to the Alphard form. Further, the syntax macro concept reported in TRW 1975 IR&D [5], e.g.

<statement> :: = IN <expression> <statement>

where this extension augments the operation of the parser, is a more user-oriented approach than reported in the literature [6]. Previous syntax macro ideas allow much more detailed user control (e.g. character scan, etc.), and yet do not seem to provide benefits valuable to non-compiler-writer users.

A review of the TRW CPDL IR&D effort [7,8,9] to determine applicability to SMITE and SMITE extensibility shows the developing syntax and semantics to be workable as a hardware connection description language. For instance, in CPDL, one might write

```
PROCESSOR:  processor-2
   (other attributes)
   SUBSTRUCTURE:  (a link to a SMITE description)
```

This will allow CPDL to provide the necessary connection

language power for SMITE, and also implies that simplified requirements (i.e. no attempt to handle connections) may be levied on SMITE extensibility.

A trial Alphard-like definition of an associative memory is as follows:

```
ASSOCIATIVE-MEMORY: DEVICE;
   Syntax Specification
      <data item> :=  ID [1] <t> <d> ASSOCIATIVE-MEMORY;
      <primitive> :=  ID [t] <extract>
   Representation (Unique)
      DECLARE S[1;L]  REGISTER,
         MWORD <1:t+d> DATA,
         MTAG<1:t> DATA,DEFINED MWORD<1:t>,
         MDATA<1:d> DATA DEFINED MWORD <t+1:d>,
         I<???> REGISTER;
   Implementation
      <-  (ADDRESS, TAG, DATA)
         S[ADDRESS]. MTAG <-t;
         S[ADDRESS]. MDATA<-d;
         or
         S[]<-t//d
      Read
         Search:  DO FOR IC - 1 to 1;
            IF t = S[I].MTAG
               THEN ESCAPE SEARCH;
               END IF;
         RETURN  S[I].MDATA
   ASSOCIATIVE-MEMORY: END DEVICE;
```

The usage might look like

```
DECLARE MAP[16] <18><32> ASSOCIATIVE-MEMORY;
```

For a t<18> d<32> memory of 16 cells.

```
MAP <- (ADDRESS, TAG, DATA)     stores,

--- <- Read(MAP[TAG])                  reads.
```

It seems all cases of forms (perhaps called a Device in SMITE) will declare new storage class attributes, in this case ASSOCIATIVE-MEMORY. We'll also need the capability to change existing ones - e.g., to re-define existing operators which are (poorly) defined for REGISTER, PORT, etc.

The section Syntax Specification is poorly integrated into everything, and is clearly incomplete as it fails to handle

the case of store. What is really desired here is an abstract specification of what an associative memory does. For now, a text block for narrative description will have to suffice.

Other than certain awkwardness, the representation section has a problem with the declaration of I - we need to know how many bits are required in an address (counter) register, which in this case is equivalent to WIDTH(16)=4.

There is also a problem with the usage section: MAP <- address, tag, data is poor syntax.

The essential needed concepts shown in this trial seem to be:

1. Group the abstractions/operations for a device together.

2. Specify the syntax involved.

3. Specify the correct representation.

4. Specify the function performed/the nature of the device.

5. Specify the implementation of allowed operations.

6. Declaration implies the device (ASSOCIATIVE-MEMORY) is a storage class attribute.

From reading the material on SIMULA [10] and Alphard [6] the following observations may be made:

1. The form/class concatenation idea is essential for reasonably compact use of extensibility.

2. Alphard presents only loop control (iteration - like) user extensibility - the fundamental loop syntax forms are fixed. An argument can probably be constructed for SMITE such that new statements are either disguised function calls (the degenerate case) or else a syntactic shorthand for a sequence of SMITE statements. This would imply fairly severe restrictions on what users could do in the way of controlling context block creation.

3. Nothing in Alphard considers the problem of asynchronous control. Advanced SMITE will provide no capability beyond that of PARALLEL-BEGIN/PARALLEL-END, although it's clear some tasking primitives such as synchronization and dispatching are required. For example,

a small but relatively complete set of operating system functions could include dispatch, recall, and wait, with meanings as follows:

dispatch - initiates the execution of a task.

recall - momentarily suspends a task. Re-activation occurs the next time the scheduler advances to the task.

wait - The task is suspended until a condition is posted.

A task termination request is also required, as is an efficient means of terminating tasks at PARALLEL-END to control when the main line continues and to determine the continuation set of registers.

There is no current need for control of a task during execution by another task (e.g., abnormal termination, but this needs thought relevant to needs for extended ESCAPE (out of a PROCESSOR).

1.2 The Definition of New Statements

One of the initial issues in SMITE extensibility is the degree to which new statements may be added to SMITE.

Examination of the published Alphard material [6] indicates that control-related abstraction in Alphard is solely concerned with extension of the types of looping constructs available. Other forms are not discussed - e.g., forms of CASE-like decoding, interrupt-handling abstractions, abnormal procedure terminations, etc.

One position would be to take the "extreme-macro" position - nothing can be introduced into SMITE which could not be built from the base-language set of control structures. This approach is more primitive than Alphard generators, even, but does support useful extensions. For example, a sparse decode could (laboriously) be represented as:

```
DECODE  opcode;
   0'000',0'001',0'002': -----
   0'472',0'702': -----
   OTHERWISE: -----
   END DECODE;
```

There would be a requirement to decode, sort, and interpret the opcodes at "macro-expansion" time if a transformation to

the emulation-efficient SMITE CASE was desired. Further, it would not provide any good way to handle the use of don't care bits in certain opcodes. For example, O'7X3' might mean that the middle 3 bits could be anything - 0 to 7.

Suppose a common operation structure is to pulse a bus, check status, and perform one of three responses. A "nice" way of writing this might be:

```
BUS FUNCTION:  A
   WHEN s1  B,
   WHEN s2  C,
   WHEN S3,  D,
   OTHERWISE  E;
```

Where s1, s2, s3 are the "well-known" status responses. The essential restrictions SMITE would impose on such a scheme would be that they would have to be structured (1 in, 1 out forms), and have to obey the established restrictions about ESCAPES past IN, PARALLEL.

The inability to generate new patterns of control is a restriction needing consideration. For instance, it is reasonably clear that a "master clear" or even an abort from deep inside a memory map box causes complete alteration of control in non-structured ways (although decidedly finite ways - usually some registers get initialized and interpretation begins anew). In SMITE this implies the need for ESCAPE to outside the current processor. Although probably to a lexically nested point. That form of control structure extension is not available with macro extensibility.

The key property (from the standpoint of the compiler) we must retain is reducibility of control graphs (nesting even). The entire code generation and optimization structure assumes that property.

It is not clear that all possible/desirable additions to the language should be through extensibility. The generality required to reach that goal might produce a scheme usable only to its developers.

Further consideration of the associative memory example shows the following view:

1. We need to add data types and operators.

2. We need to be able to add operators to existing data types.

3. We need to be able to add new statements.

It seems that the full power of syntax macros is too cumbersome for simply adding data types and operators - instead, the compiler could perform automatic syntax extension.

It is possible to write an improved version of the associative memory using the automatic extension syntax idea:

```
ASSOCIATIVE-MEMORY:  DEVICE [1] <t,d>;
   Specification
      "The associative memory supports the storage and
      retrieval of data items by a tag value used as an
      address."
   Representation
      DECLARE
         S[1:1] REGISTER,
         MWORD <1:t+d> DATA,
            . . .
         I<WIDTH(1):1> REGISTER,
   Implementation
      operator <-
         operands address, tag, data : word
         returns value : word
            value <- s[address].mtag <- tag;
               s[address].mdata <- data;
      operator read
         operands tag :  word
         returns value : word
         SEARCH:  DO FOR I <- 1 to 1;
            IF t = s[I].mtag
            THEN ESCAPE SEARCH;
            END IF;
            value <- s[I].mtag
   ASSOCIATIVE-MEMORY:END DEVICE;
```

Given the assumption that the first operand for operators on a subscripted device has to be the address (could use * to indicate not used), then references would be if the form

```
DECLARE Y[512]<10,17> ASSOCIATIVE-MEMORY;
. . .
Y[1] <- X'3F', IN-PORT;
OUT-PORT <- read(Y[TAG-VALUE]) ;
```

where the convention is that multiple operands are strung to the right of an infix operator and are separated by commas.

The example also demonstrates the need for a more primitive data type in SMITE than REGISTER, etc. Call it the WORD (i.e., bit string). Then, in principle, all the present data types in SMITE can be defined as extensions of WORD. For example,

```
PORT:DEVICE <lb:ub> EXTENDS WORD;
   Specification  "A PORT is an externally accessible
   register."
   Representation  (none)  (But need compile-time ordinal)
   Implementation
      operator <-
         operands value: word
         returns value:  word
            DIRECT;
               "setup value in R0"
               LDI    R.ADR,ordinal
               SYSTEM USR.RCL+1
               SMITE;
   PORT: END DEVICE;
```

and so forth. We do need some mechanism to deal with devices which may or may not be subscripted.

1.3 Domain Coupling

One of the semantic issues in the definition of extensibility in SMITE is open or closed procedure insertion. Although of concern in a purely sequential environment, where inline macro expansion can result in a call-by-name effect, but closed expansion results in an effect determined by the means of parameter passing chosen, the inline/closed expansion issue is even more significant in a concurrent execution environment due to asynchronous coupling of references to shared variables.

The core of the problem is the question of communication and synchronization between separate execution domains. In an ALGOL-like sequential world, where we are passing parameters via CBN(call-by-name), such as

```
   . . .
   F(A(J),J,A(J+2))
   . . .
where
   PROCEDURE F(A,B,C)
      . . .
      X=A
      B=B-2
```

```
Y=C
. . .
```

The result after the assignment to Y is that Y holds identically the value in cell A(J) (J is evaluated before the call) as does X. What happened during execution may be visualized as F reaching into the execution domain of the calling procedure through its parameters to manipulate variables. Consideration of "thunk" techniques for implementation of CBN reveals a direct use of the concept: every time an evaluation of the parameter is made, a call is executed to a thunk back in the calling domain. The parameters retain scoping and definition local to the calling procedure.

The current implementation and definition of SMITE does not allow this blurring of execution domains. In SMITE, parameters are passed by call-by-value or call-by-value-returned. Any alteration of values in the calling domain is performed only after the called processor returns control. During execution of the called processor, parameters are strictly local to the called processor.

Considerations of extensibility and concurrency to be implemented in Advanced SMITE require us to extend this thinking. Consider extensibility first.

1.3.1 Domain Coupling under Extensibility.

The same form textually expanded either in-line or in closed form, gets different linkages to the calling domain. With inline expansion, the form is completely merged into the calling domain - no separate called domain exists. With closed expansion, a separate called domain is formed. This effect is unsatisfactory - the method of expansion, which has only vague analogs in hardware, should not produce changes in the action invoked.

One possible solution is to mimic the SMITE CBV (call-by-value) and CBVR (call-by-value-returned) actions in the expansion of inline processor calls on forms. Parameters would be restricted to words - that is the inputs to a function of a form are words. Form parameters (i.e., when the instance of the form is declared) might not require that restriction.

The word restriction leads to the issue of coupling asynchronous domains. Interaction between asynchronous domains implies synchronization and cooperation - something

implicitly provided in a simplistic manner by parameters. Rather than overburden a basically simple, desirable mechanism with a lot of conventions to handle complex interactions, another alternative is available in the extension of the PORT concept.

1.3.2 Domain Coupling and Concurrency.

A means of coupling the domains of asynchronous (concurrent) processes is required. Look at these two examples: Paged CPU (Figure 2) and PDP-11 (Figure 3).

In the Paged CPU example, all the connections could be "simple" - i.e., the connection is as simple as some wires, with no "intelligence" behind it. Synchronization is embodied in the units - no formal line protocol is required. Another example would be the connections to an ALU. (Some implementations of the pager might involve active intelligence in the pager - for this example imagine that it is transparent in the sense that the communications between CPU and P are the same as P and M, or as CPU and M would be if the system had no pager installed.)

This example can be handled using shared variables and processor calls where required. Since no active asynchrony exists, it is well within the capability of what is foreseen for Advanced SMITE, without the need for the connection language. For example, in the language extension, we might write forms for associative memory (as described earlier) and the pager. The pager form might be like this:

```
Pager:  DEVICE(a1,a2,a3,a4 : ASSOCIATIVE-MEMORY)
<lb:ub>;
   Specification
      "The paging device maps memory addresses from the
      CPU into physical addresses sent to real memory.
      The associative memory used is selected based on
      the program status."
   Representation
      Declare Global  A-M-SELECT<0:1> DEFINED
      PSW<20:21>;
   Implementation
      OPERATOR READ
         OPERANDS address:word
         RETURNS value<lb:ub>:word
         CASE A-M-SELECT;
            value <- a1[address<0:1>]//address<2:1>
            value <- a2[address<0:1>]//address<2:1>
```

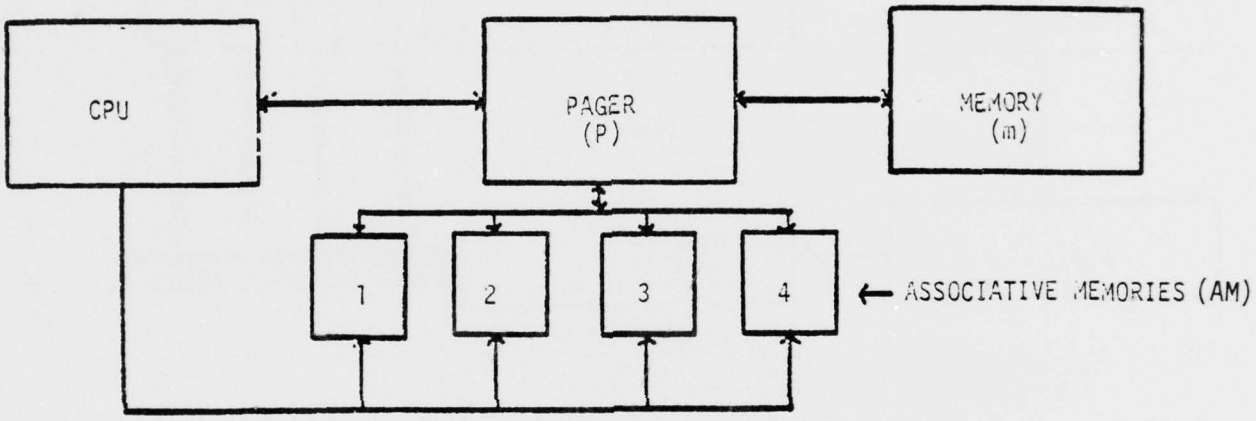


Figure 2

Paged CPU

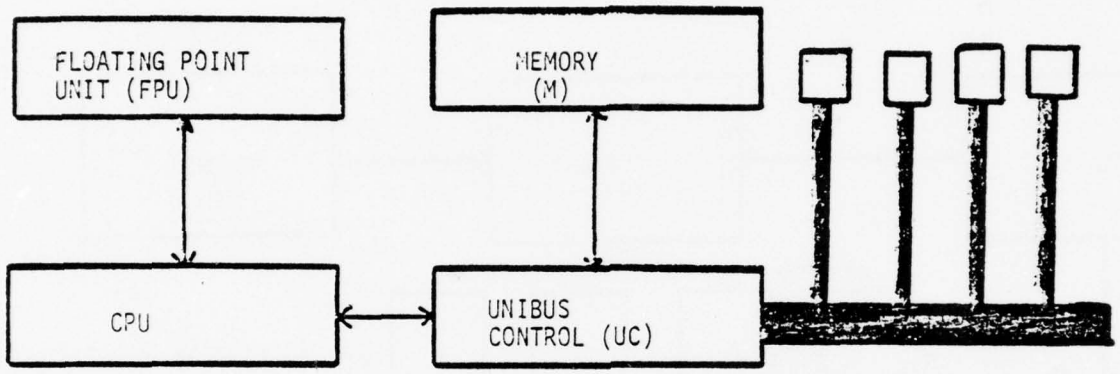


Figure 3

PDP-11

```
                value <- a3[address<0:1>]//address<2:1>
                value <- a4[address<0:1>]//address<2:1>
                END CASE;
      Pager:  END DEVICE;
```

The third form might be a paged memory.

```
   Paged-Memory: DEVICE (p:pager) [1] <lb:ub>;
      Specification
         "Paged memory is essentially the same as normal
         memory except that all address references to it
         are translated through the pager."
      Representation
         DECLARE M[0:1-1]<lb:ub>;
      Implementation
         OPERATOR READ
            OPERANDS address: word;
            RETURNS value:word;
            value <- M[p(address)];
         OPERATOR <-
            OPERANDS address, valuein: word;
            RETURNS valueout: word;
            valueout <- M[p(address)] <- valuein;
      Paged-Memory:  END DEVICE;
```

The declaration of a CPU becomes clear and simple using these three forms. There is a loose end with respect to the GLOBAL in the pager - perhaps it should be forced to be a parameter to the declaration.

```
   CPU:PROCESSOR;
      DECLARE
         PSW<0:31> REGISTER,
         AM1[4]<2,12> ASSOCIATIVE-MEMORY,
         AM2[4]<2,12> ASSOCIATIVE-MEMORY,
         AM3[4]<2,12> ASSOCIATIVE-MEMORY,
         AM4[4]<2,12> ASSOCIATIVE-MEMORY,
         P(AM1,AM2,AM3,AM4)<0:33> PAGER,
         M(P)[0'100000000000']<0:31> PAGED-MEMORY;
      . . .
      IR <-  READ(M[PC]) ;
      PC <- PC+1;
      .
      etc.
```

A means is required to setup arrays of forms - e.g., an array of associative memories.

The second example, the PDP-11, contains this form of

connection from FPU-CPU, CPU-UC, UC-M. The connection language should support description of these passive connections as well as descriptions like given above. The PDP-11 also contains active connections, shown with the heavier lines, which are in fact the unibus.

In the unibus connections, active (intelligent) devices are connected to each other, and a complete synchronization protocol exists. Contrast this to the interface of a processor to a memory, where the interface is simply request, data lines, and busy/data available. (This discussion is artificial in some senses, since the differences are not absolute as the terms active and passive would suggest, but are in fact only of degree).

For "active" connections, the PORT-type of mechanism is appropriate. A device defined either similarly to a PORT or even extending PORT could then incorporate both the formal protocols used over the connection, and the means to implement on the QM-1. In the PDP-11, a UNIBUS-CONTROLLER processor, which itself could be a DEVICE in the PDP-11 description would have a DEVICE UNIBUS which would be the point of contact with the connection language.

1.4 Conclusion

The concept of an extensibility mechanism based on the Alphard 'form' appears to be a viable concept for use within the SMITE language. Coupled with the use of syntax macros this mechanism provides a method of defining new data types (DEVICEs) and the functions necessary to support those data types. The definition of the new types affords a higher degree of abstraction than previously available in the SMITE language.

2.0 Analysis of Syntax Macros in Advanced SMITE

This section contains the working notes compiled during the study of syntax macros and their use within Advanced SMITE. This study was concerned with the following tasks:

1. Defining the user accessible grammer, simplified and restricted from the full grammer accepted by the parser.

2. Determining the necessary constructs required for the definition of syntax macros.

3. Determining the mechanism required for the expansion of syntax macros.

2.1 Introduction

The syntax macro is a method of extension that allows the user to alter the actual syntax of the language. Since the syntax macro allows alteration of the syntax of the language the user must be aware of the base syntax and any additions previously made when defining any additional changes. For this and other reasons it has been decided early in this study to use syntax macros only for changing control structures (statements and primitives) and to adopt a version of the ALPHARD form (Device) for creating data and operator extensions. Using the Device allows the use of an "automatic" syntax extension mechanism associated with the Device. It is felt that this decision is appropriate because the Device is powerful enough and more straight forward in describing data and operator extensions. For example the syntax macro definition of a stack type might be:

```
MACRO "A<ID> STACK [B<CONSTANT>] C<DECTYPE>" EXTENDS
<DECPHRASE>;
   MEANS
      DECLARE $A[1:B] C, $P <0:35> REGISTER;
      MACRO "SIZE-OF A" EXTENDS <IDPRIMITIVE>,
         MEANS
            $P;
         MACRO "A" EXTENDS <IDPRIMITIVE>,
            MEANS
               IF $P <= 0
                  THEN ERROR ($P);
                  ELSE BEGIN;
                     $A[$P];
                     $P <- $P - 1;
                     END;
                  ENDIF;
            MACRO "A <- D<EXPRESSION>" EXTENDS <EXPRESSION>,
               MEANS
                  IF $P > B
                     THEN ERROR($P);
                     ELSE BEGIN;
                        $P <- $P + 1;
                        $A[$P] <- D;
                        END;
                     ENDIF;
            MACRO " ERROR E<PROCCLL>;" EXTENDS<STATEMENT>,
               MEANS
               (SOME SORT OF ERROR PROCESSING);
```

while the form definition might be:

```
STACK:DEVICE[1] type EXTENDS WORD;
   Specification "The stack type implements the
      push and pop stack function and
      also allows the user to determine
      the current size of the stack"
   Representation:  DECLARE
      S[1:1] TYPE,
      P <0:35> REGISTER;
   Implementation:
      operator <-
         operands   . . .
         returns    . . .
            IF P > 1 THEN ERROR(P);
               ELSE BEGIN;
                  P <- P+1;
                  S.[P] <- VALUE;
                  END
               ENDIF;
      operator pop
         operands   . . .
         returns    . . .
            IF P <= 0 THEN ERROR(P);
               ELSE BEGIN;
                  value <- S[P];
                  P <- P - 1;
                     END;
               ENDIF;
      operator "SIZE OF"
         operands   . . .
         returns    . . .
            value <- P;
   STACK: END DEVICE;
```

The form definition seems much cleaner and does not force the user to know the syntactic entities for the extensions. Therefore the only use for syntax macros will probably be the extension of control structures.

Syntax macros should be used only to create alternative definitions for (extend) basic SMITE syntactic entities. They should not be used to redefine existing syntactic entities, which would alter the base SMITE language. Also, syntax macros should not be used to create entirely new entities, which would amount to defining new non-terminals within the formal definition of the language. The ability to restructure the base language is definitely not needed to achieve the goals we have defined for syntax macros.

The user should not have access to all the compiler syntactic entities because it would be quite confusing to decide which entity to actually extend. Therefore, only the syntactic entities STATEMENT and PRIMITIVE will be extendable. There is no reason to go deeper into the workings of the compiler to accomplish the capability that syntax macros are to provide.

The information that must now be determined is to which syntactic entities (non-terminals in the grammer) the user must have access to allow him to extend the entities STATEMENT and PRIMITIVE.

2.2 Definition

There must be a method of introducing the syntax macro into the base language. One idea is that a syntax macro could look like a declaration and appear any where a declaration may appear. Another approach is that all the extensions would be an entity separate from the actual SMITE description and pre-processed before the actual compilation begins. The usefulness of a pre-processor for syntax macros is limited by the amount of semantics present in the syntax macro, which will also dictate the type of pre-processing that can be done. In either case the actual definition needs the same basic items. A method of indicating what the macro call actually looks like, what syntactic entity the macro expands, the implementation of the macro in the existing language and any semantic notions needed to resolve ambiguities caused by the extension.

It appears that a syntax macro can be used to extend data types and operators but it would be more useful to use DEVICES because the user is relieved of the burden of having to explicitly extend the syntax. He could concentrate on the DEVICE itself and let the compiler extend the syntax. The use of the DEVICE in declaring a stack type also seems to be clearer:

```
STACK:device [1] extends word;
```

instead of

```
macro "A<ID> STACK [B<CONSTANT>] extends <DECPHRASE>";
```

Also, using the DEVICE the user does not have to explicitly know the syntactic entities that need extension.

Syntax macros however are needed to add new statements. The

question is how much of the base language syntax need be known to allow the expression the information required.

Adding semaphores for the synchronization of concurrent processes is a good example of statements that would be desirable to add.

```
Macro "SIGNAL A<SEMAPHORE>;"extends <STATEMENT>,
   INTERNAL.A<-INTERNAL.A+1
   if INTERNAL.A <= 0(direct code to wake up a waiting
   process)
   macro end;
macro "WAIT A<SEMAPHORE>;" extends <STATEMENT>,
   INTERNAL.A<-INTERNAL.A-1;
   if INTERNAL.A < 0(direct code to suspend a process)
   macro end;
```

and finally:

```
SEMAPHORE:Device Extends Word;
   Declare INTERNAL Register;
   semaphore description and semantics;
   end device;
```

Note how the SIGNAL and WAIT statements look almost like processor calls except without the parenthesis. This is one of the uses of a syntax macro, allowing the user defined syntax of "processor calls". It could be left up to the compiler to decide if the resulting code was in-line or actually treated as a call.

Processing of syntax macro calls consists of three operations:

1. Recognizing the macro call

2. Verification of the required semantics

3. Generation of the replacement.

Following is an example of the description of a cache memory using the concepts introduced and discussed so far.

```
CACHE-MEMORY:DEVICE [lo:l1]<lb:ub>;
   SPECIFICATION:
      "THIS IS A CACHE MEMORY EXTENSION IN WHICH THE CACHE
      IS OF SIZE 8 AND USES THE LIFO METHOD OF SATISFYING
      MEMORY ACCESSES"
      OPERATORS ARE <-get-value;
      MACRO IS IN-CACHE;
```

```
REPRESENTATION:
   DECLARE UNIQUE CACHE-ADD[0:7]<0:15>FAST-MEMORY,
   CACHE [0:7]<lb:ub>FAST-MEMORY,
   MEM[lo:11]<lb:ub>MEMORY,
   PNT<0:2>FAST-MEMORY;
IMPLEMENTATION
   OPERATOR <-      operands a,b:word;
      result value:word;
      CACHE-ADD[PNT] <- b;
      value <- MEM[b] <- CACHE[PNT] <- a;
      PNT <- PNT+1;
   OPERATOR get-value       operand b:word;
      result value:word;
      LOOK :BEGIN;
         TEMP <- PNT-1;
         DO FOR 0 to 7;
            IF CACHE-ADD[TEMP]=b THEN
            BEGIN;
               value <- CACHE[TEMP];
               ESCAPE LOOK;
               END;
            ENDIF;
         value <- CACHE [PNT] <- MEM[b];
         CACHE <- ADD[PNT] <- b;
         PNT <- PNT+1;
         LOOK:END;
   IN-CACHE:MACRO"A<EXP> IS IN CACHE OF B<ID>";
      WHERE
         TYPE(B) = CACHE-MEMORY,
         TEMPORARY I<0:2> REGISTER;
      MEANS
         C:BEGIN;
            DO FOR I <- 0 TO 7;
               IF CACHE-ADD[I] =A THEN
                  BEGIN;
                  RETURNS 1;
                  ESCAPE C;
                  END;
                  ENDIF;
            RETURNS 0;
            C:END;
         IN-CACHE:END MACRO;
```

The IN-CACHE extension brings up a problem. The macro needed an internal and unique ID to do the looking through the array CACHE-ADDR. The needed ID was declared in the representation of the device.

A syntax macro should be composed of 3 parts. The first part

will contain the syntax of the macro call (probably called the structure clause). The second part will contain any needed semantics (perhaps the where clause). The third part will actually describe the implementation of the macro and could be called the means clause or definition clause. The syntax macro syntax could look like:

```
MACRO <structure clause>

WHERE <where clause>

MEANS <means clause>

END MACRO;
```

The structure clause needs only to specify the syntax of the macro call i.e.,

```
MACRO "FOR ALL A<EXP> IN C<ID>;"
```

Note also that any context-sensitive requirements are also specified in the structure and all the parameters (which are non-terminals in the base language) must be identified by their non-terminal names. The other tokens that appear in the structure clause are considered terminals and are addded to the reserved word list if not already there. No indication need be made to determine if the macro extends STATEMENT or PRIMITIVE because in the means clause the existence of a returns statement (indicating the value to be returned) will indicate an extension of PRIMITIVE otherwise it is an extension of STATEMENT. The only non-terminals the user can use in a syntax macro are ID and EXPRESSION. These entities appear to be sufficient to allow complete definitions of new STATEMENTS AND PRIMITIVES.

The where clause of a syntax macro provides the necessary semantic information. The information consists basically of specifying particular values for various attributes of the parameters (non-terminals) of the syntax macro. The attributes for the non-terminals that need to be accessed are:

```
ID : width,type (capabilities),length

EXPRESSION:width,type (capabilities)
```

The where clause must provide a method of specifying the various requirements on the parameters of the macro. In addition to the type, there must also be a method of specifying the further requirement of capabilities, i.e.,

verify that certain operators or functions are defined concerning the specific type required.

All requirements of the where clause must be met for the STATEMENT or PRIMITIVE to be considered a legal syntactic construct, not to mention that the structure clause must be matched. If the structure clause is met but the where clause is not met, this does not necessarily mean that the code is illegal, since there may be another syntax macro which has requirements the code meets.

The means clause is straightforward. It is merely a description of the actual implementation of the syntax macro. The description should be able to use all the constructs of the base language plus any extensions added.

All parameters into the macro (as defined in the structure clause) can be used within the means clause anywhere it is legal to use that syntactic type.

To provide for the syntax macro definition mechanism described here, the following statements must be added to SMITE:

MACRO - the overall statement for defining syntax macros. Its parts will consist of the following:

a structure clause which describes the syntax of the macro call and formal parameters.

a where clause which describes any semantics necessary to further qualify the macro call.

a means clause which provides the actual implementation of the macro plus the parameters provided by the structure and where clauses.

RETURNS - a new construct that is legal only within the means clause of a syntax macro and signifies that the syntax macro is actually on extension of PRIMITIVE. Absence of the RETURNS statement implies that the syntax macro extends STATEMENT.

The actual call of a syntax macro is handled as follows:

The compiler first deduces that what it is currently processing is a possible syntax macro. This is achieved by building parser tables used from the information in the structure clause. The where clause is then examined and all requirements are checked. If the test is passed the

means clause is expanded. The result of the expansion must then be parsed and if it contains any macro calls, they will be handled in the same manner. The context of any expansion must be unique because the means clause can contain labels and consequently context blocks.

The syntax macro can be used in two roles. One to add any constructs necessary to support an abstraction added by a device. The other role is to add general constructs. In the latter, there might be a device that consists only of several syntax macros and no data or operator abstractions. One benefit of the former use is to allow the syntax macro to use constructs local to the device in performing its duty. This also brings up the fact that it is necessary for the context of each instance of an abstraction described by a device to be retained so that it can be used by the expansion of a related syntax macro. In fact, when the expansion of the syntax macro is performed, the correct context must be found by identifying which instance of the device the particular call of the syntax macro is referencing.

Following is a counter extension using syntax macros:

```
REG-COUNTER :DEVICE <lb:ub> EXTENDS REGISTER;
   SPECIFICATION:  "THIS IS A REGISTER BASE COUNTER"
      USES REGISTER;
      MACROS ARE INC,DEC;
   REPRESENTATION
      DECLARE I<lb:ub> REGISTER;
   IMPLEMENTATION
      INC:MACRO "INC THING<ID>"
         WHERE TYPE OF THING IS COUNTER;
         MEANS
            THING.I <- THING.I + 1;
            RETURNS THING.I;
         INC:ENDMACRO;
      DEC:MACRO "DEC Q<ID>"
         WHERE TYPE (Q)=COUNTER;
         MEANS
            Q.I <- Q.I + 1;
            RETURNS Q.I;
         DEC :ENDMACRO;
   REG-COUNTER : END DEVICE;
```

Summation macros for registers and vectors could be defined as follows:

```
SUMMERS:  DEVICE;
```

```
SPECIFICATION   "THIS DEVICE DEFINES SUMMATION EXTENSION
FOR TYPES REGISTERS AND VECTORS"
   MACROS ARE SUM-REG, SUM-VEC;
   USES REGISTER, VECTOR;
IMPLEMENTATION;
   SUM-REG:MACRO "SUM A<ID> FROM C<EXPRESSION> TO
   D<EXPRESSION>"
      WHERE
         TYPE(A) = REGISTER,
         LENGTH(A) > 0;
      MEANS
         DECLARE
            INDEX <1:MAX(WIDTH(C), WIDTH(D))> REGISTER,
            TEMPORARY I<1:WIDTH(A)> REGISTER;
         I <- 0;
         DO FOR INDEX <- C TO D;
            I <-  I + A[INDEX];
         RETURNS I;
      SUM-REG:  END MACRO
   SUM-VEC :  MACRO "SUM A<ID> FROM C<EXP> TO D<EXP>"
      WHERE
         TYPE (A) = VECTOR,
         LENGTH(A) > 0;
      MEANS
         DECLARE
            INDEX <1:MAX(WIDTH(C), WIDTH(D))>,
            TEMPORARY I<1:WIDTH(A)> VECTOR[SIZE(A)];
         I <- 0;
         DO FOR INDEX <- C TO D;
            I <- I + A[INDEX];
         RETURNS I;
      SUM-VEC: ENDMACRO;
SUMMERS:END DEVICE;
```

The summers could be used as follows:

```
DECLARE A<0:3>[1:10]REGISTER;
DECLARE V<15:0>[0:31]VECTOR [10];
   . . .
CASE   SUM A[C] USING C <- FROM 4 to 19;
   . . .
Q <- B*SUM V[D] USING D <- FROM NEXT-1 TO 31;
```

The above example some points:

1) There also seems to be a problem in the declaration of the temporary, I. The width (length) of the type needs to be known. (Note that the actual value is no problem if a ZERO function is defined for the type). We may have to go

so far as to indicate the bit ordering and numbering and do the same for length, i.e.

TEMPORARY I<LW(A):RW(A)>,

where LW = left width

RW = right width

The following would also be used:

LL : LEFT LENGTH

RL: RIGHT LENGTH

The Means clause appears to be a suitable location for the declarations of identifiers need by a syntax macro. In this manner, a device that is used only to provide syntax macros there need not be any declarations in the REPRESENTATION clause and therefore that device need not be DECLARED in the description (other than in the USES clause).

2) The FOR loop in both summations assumes integral iteration. If a different type of iteration was desired it would be necessary to write the MEANS clause using a different iterator.

3) There is another problem in the declaration of the temporary for the SUM-VEC syntax macro. It would be desirable to declare the temporary vector the same size as the summed vector but "SAME SIZE" does not seem to be the correct way to do it, in fact the real problem here is the method of referring to attributes that are results of extensions. These attributes can be accessed as described in 2) above or by referring directly to the parameters of the device (which will be parameters of the instantiation of the specific data item in question).

These summation syntax macros bring up another aspect. For example, suppose that the "+" operator was not closed over the type VECTOR. This causes no problem because the syntax macro author would have to know what type of the result of VECTOR "+" and that would be the type of the source variable. All users of the VECTOR type must be aware of all ramifications and interfaces of the type.

If a device is going to use other abstractions it must declare which abstractions it requires (with the USES statement).

Even the "SUMMERS" device, which is only used to add statements, must declare any abstractions it needs to support its internal workings.

No operator or syntax macro defined within a device can be used elsewhere within the same device. This is logical because the syntax macro and operator are extensions designed to work upon the abstraction defined by the device.

The device can also contain helper processors to aid in definition of the device. These processors are standard SMITE processors with the exception that they are only defined within the context of the device.

Any syntax macro that is an extension of statement cannot have its invocation labeled, i.e.

```
A:  MACRO "SIGNAL A<ID>"
    . . .
    (MEANS clause does not contain a RETURNS)
    A:  END MACRO
    . . .
LABEL:  SIGNAL Q;
```

The label on the SIGNAL statement is illegal because a syntax macro may not be labeled. Within the syntax macro the author can create context blocks, but the syntax macro mechanism itself does not. Also any context blocks created within the syntax macro cannot be referred to by the calling program.

In the syntax macro WHERE clause the following is to be used to force any semantic requirement on syntax macro parameters:

```
MACRO "FOR ALL A<EXP> IN B<ID> INCREMENT"
   WHERE B:  ASSOCIATIVE-MEMORY,
```

The above WHERE clause allows the user to declare type and capability requirements. The WHERE clause can also specify WIDTH and LENGTH requirements on syntax macro parameters, such as:

```
WHERE WIDTH(A) = 4, LENGTH (B) < = 23;
```

Also the MIN and MAX functions can be used within the WHERE:

```
WHERE MAX (WIDTH(A), WIDTH(D)) < 26;
```

The LENGTH function can also be used just to specify the parameter has the LENGTH attribute:

```
   WHERE LENGTH(A);
```

WIDTH, LENGTH, MIN and MAX can all be used in the declarations of the MEANS clause. They can essentially be thought of as compile time constants. In fact they can be used anywhere in the MEANS statement where a constant is appropriate.

Another example of the use of semantic attributes is:

```
   ...

   WHERE

      A<W1:W2>[L]:VECTOR

   MEANS

      DECLARE I<1:WIDTH(A)>VECTOR[L];

      ...
```

In addition to requiring A to be a vector, the left and right bit numbers of the parameter and the square bracket attribute (vector-length) are labeled. The data item I could also be declared as follows:

```
   DECLARE I<W1:W2>[L]:VECTOR;
```

or as

```
   DECLARE I<A.W1:A.W2>[L]:VECTOR;
```

If the SUM-VEC syntax macro appeared in the following VECTOR extension:

```
   VECTOR:DEVICE[1]<1u:rw> EXTENDS WORD;
      SPEC "--" . USES --;
      REPRESENTATION DECLARE J[1:1]<1w:rw>;
      IMPLEMENTATION
         ...
```

The following could have appeared in the WHERE clause and MEANS clause:

```
   ...

   WHERE A :VECTOR;
   MEANS
   DECLARE I<1w:rw>[1]VECTOR;

   ...
```

This is because the macro is within the device and has access to all information the device has access to. No qualification was needed, because only one vector (A) is present, and the lw, rw and l can be ascertained from the A instance of VECTOR.

If the following macro:

```
MACRO ' -- A<ID> --B<ID>
   WHERE A:VECTOR,
      B:VECTOR;
   MEANS
   DECLARE I<lw:rw>VECTOR [l],
   J<lw:rw>VECTOR [l];
```

is within the same VECTOR device just described, then the DECLARE statement is ambiguous, because it is not known which lw, rw or l to use, the ones from the A instance or the ones from the B instance.

However, if they are qualified such as:

```
DECLARE I<A.lw:B.rw>VECTOR [a.l]
J<B.lw:B.rw>VECTOR [B.l];
```

everything works correctly.

The same problem applies when referring to attributes described in the WHERE clause, for example:

```
WHERE A<w1:w2>[l]:VECTOR;
B<w1:w2>[l]:VECTOR;
```

The qualification mechanism is also needed to refer to the correct w1, w2 and l. The need for qualification can be eliminated by unique names such as:

```
WHERE A<w1:w2>[10]:VECTOR;
B<w3:W4>[11]:VECTOR:
```

Qualification is needed (and required) within a device only where names don't uniquely identify the desired quantity (when referring to concrete data items of an instance when there is more than one instance of the same device in the same context).

Following is a decimal extension:

```
DEC-DIG:DEVICE(C-FLAG:FLAG)EXTENDS WORD { <- };
   SPECIFICATION "this is a decimal digit"
      operators are +, -, <- ;
      macro is IS-DD;
      uses REGISTER, FLAG;
   IMPLEMENTATION
      operator + operands a,b:DEC-DIG:
         result c:DEC-DIG;
         C-FLAG//c.I <- ADD-EM(a.I,b.I,C-FLAG);
      operator - operands a,b:DEC-DIG;
         result c:DEC-DIG;
         DECLARE TEMP<3:0>REGISTER;
         IF b.I = 0 THEN TEMP <- 0;
            ELSE TEMP <- B'1010' - b.I;
            ENDIF;
         C-FLAG//c.I <- ADD-EM(a.I,TEMP,C-FLAG);
      ADD-EM:PROCESSOR<4:0>(A,B,C);
         DECLARE A<3:0>REGISTER, B<3:0>REGISTER, C FLAG;
         ADD-EM <- 0//A + B + C;
         IF ADD-EM >9 THEN ADD-EM <- ADD-EM + 6;
            ENDIF;
         ADD-EM:END PROCESSOR;
      IS-DD:MACRO "IS A<ID> DEC-DIG"
         WHERE A:DEC-DIG;
         MEANS IF A<10 THEN 1 ELSE 0;
         IS-DD:END MACRO;
   DEC-DIG:END DEVICE;

DEC-WORD:DEVICE[1](C-FLAG:FLAG);
   SPECIFICATION
      "DECIMAL WORD OF 1 DECIMAL DIGITS"
      operators are +, -, <- ;
      uses DEC-DIG,REGISTER;
      macro IS-DW;
   REPRESENTATION
      DECLARE J[1:1]DEC-DIG(C-FLAG);
   IMPLEMENTATION
      operator + operands a[1], b[1]:DEC-WORD,a.1=b.1;
         result c[a.1]:DEC-WORD;
         DECLARE I<1:WIDTH(c.1)> REGISTER;
         DO FOR I <- 1 TO c.1;
            c.J[1] <- a.J[I] + b.J[1];
      operator  -  operands a [1]],b[1]:DEC-WORD, a.1=b.1;
         result c[a.1]:DEC-WORD;
         DECLARE I <1:WIDTH(c.1)>REGISTER;
         DO FOR I <- 1 TO c.1;
            c.J[I]<- a.J[I] - b.J[I];
      operator <- operands a[1],b[1]:DEC-WORD,a.1=b.1;
         result c[b.1] :DEC-WORD;
```

```
      DECLARE I<1:WIDTH(c.1)> REGISTER;
      DO FOR I <- 1 TO b.1;
         c.J[I] <- a.J[I] <- b.J[I];
   IS-DW: MACRO " IS A<ID> DECIMAL WORD"
      WHERE
         A:DEC-WORD;
      MEANS
         DECLARE TEMP<1:WIDTH(A)> REGISTER;
         LABO:BEGIN
            DO FOR I <- 1 TO A.1;
                  NOT IS A.J[I] DEC-DIG THEN
                   _ IN;
                  RETURNS 0;
                  ESCAPE LABO;
                  E D;
                  ENDIF;
            RETURNS.1;
            LABO:END
      IS-DW:ENDMACRO;
DEC-WORD:ENDDEVICE;
```

In the above example, some mechanism is needed to describe the semantics for the operands of the operators. Also some method was needed for describing the "shape" of the results of operators. The IS-DW syntax macro refers to data in the representation clause, (A.J[I]).

The parameter C-FLAG was not exactly carried through correctly. No check was made to ensure that all items were declared using the same flag. In fact, the entire issue was side-stepped by not specifying which C-FLAG, associated with which parameter, to use in + for DEC-DIG for instance.

2.3 Conclusions

It has been determined that the only two SMITE syntactic entities that can be extended are STATEMENT and PRIMITIVE. This limited method of extension coupled with the DEVICE extension allows the extension of data types, operations and control structures, while keeping the interpretation of extensibility within reason.

A syntax macro is a separate part of the DEVICE IMPLEMENTATION clause and along with all operations defined within the clause follows the same rules of global visibility. Therefore any syntax macro that is to be seen by the outside world must be listed in the SPECIFICATION clause. Helper processors can also appear within the IMPLEMENTATION clause. The processors can never be used directly by the outside world but they can

be used indirectly through the use of a syntax macro. A syntax macro can use any other visible syntax macro or operator from other devices.

All syntax macros, helper processors and operators are separate and autonomous entities within the IMPLEMENTATION clause. Syntax macros can not be nested within one another but helper processors may have nested processors and the scope of names rule works as usual.

The entire mechanism of calling and expanding syntax macros is done at compile time. Once the syntax of the call is recognized all necessary semantic requirements are checked and the macro is expanded if so indicated. A syntax macro is a closed context block with the exception that the there is no concept of global variables, the syntax macro sees only the data local to itself and the device and any data passed through parameters. Any call of a STATEMENT syntax macro cannot be labeled since there is no method of adding new types of context blocks to the base language and there is no way of labeling a call of a PRIMITIVE syntax macro.

If the syntax macro is to be used in such a manner as to need access to the concrete data representations of a device the macro must be defined within that device and there must be a method of indicating which instance of the device is to be used.

3.0 Analysis of Direct Code in Advanced SMITE

3.1 Direct Code Extensibility

SMITE serves the dual role of a computer description language and a programming language for generating emulations. In the latter capacity, when the emulator is used primarily for analysis and execution speed is a secondary consideration, SMITE provides an ideal method for generating the emulator.

For developing emulations which have critical execution speed requirements, it is desirable to improve the efficiency of the object code produced by the SMITE compiler. This would allow efficient emulations to be produced which retain the desirable features of a SMITE computer description (structured, maintainable, descriptive, easy-to-produce).

The improved efficiency of the SMITE compiler results from

optimization and extensibility. The compiler optimization improves the efficiency of all the object code produced in a manner transparent to the SMITE programmer. The compiler extensibility adds new constructs to the language or permits microcode to be inserted directly into the computer description. The direct microcode is used to improve the efficiency of critical regions of the emulation to a degree normal optimization techniques cannot attain. The emulation execution speed can be improved substantially with a small amount of direct microcode.

Direct code extensions to the SMITE compiler can produce substantially more efficient code than an optimizing compiler for several reasons:

a. The direct code can use new QM-1 microinstructions devised for a particular application. The microinstructions may initiate parallel operations, merge ALU operations which would require several instructions, and eliminate the overhead encountered in the set up of each microinstruction. In a typical QM-1 emulation, the less critical areas are coded with the MULTI instruction set and the more critical areas are put into nanocode. The direct code extension of SMITE permits this type of optimization.

b. The SMITE compiler should not be expected in all cases to remember which registers hold expressions and avoid a penalty for reloading them. Likewise the compiler cannot infer which assignments statements in a loop can be deferred until the end of the loop. In particular, direct code extensions can improve execution speed substantially in loops or code manipulating common data.

c. The SMITE compiler uses a subset of the QM-1 Multi instruction set and possible QM-1 hardware. These instructions are not necessarily less efficient than those which are used in the compiler. In some cases (e.g., RAD, SWMS), these instructions are used for unique operations. In other instances (e.g., EXTR), the instruction as part of a code generation sequence could process only a subset of the possible operations due to width or register restrictions and a more generalized code sequence was selected. For a specific application, the unused instructions may be more efficient.

d. The SMITE compiler does not permit direct access to additional QM-1 hardware or external SMITE processors.

The first task of direct code extensibility is an analysis of the type of operations which might be used in direct code extensions. The reason for this study is to determine how much knowledge of the preceding and subsequent code must be available to the direct code and how the necessary information must be expressed.

The interface of the direct code with the SMITE description is critical and an analysis of representative cases should illustrate the potential problems. Table 1 consists of microinstructions which have been used in other emulations or are used only to a limited capacity in the compiler. The type of nanocode optimization currently being performed should give some insight into the type of direct code extensions to expect.

Table 1 Emulation Microinstructions

MULTI	
RAD	Reg + CS to Reg and CS
SW	Swap LS,CS
LDM	Load multiple
STM	Store multiple
PULL	Pop CS stack
SWMS	Swap LS and main store
EX	Execute
EXTR	Extract
SBO,SBZ	Set bit
TBO,TBZ	Test bit
LDEI	Load external store
STEI	Store external store
QM36	
LD2	36 bit CS load
ST2	36 bit CS store
LDMS2	36 bit MS load
STMS2	36 bit MS store
ADR2	36 bit add
SBR2	36 bit subtract
36 bit shifts	
36 bit logical operations	
7094	
ALU2	36 bit ALUX
GETADD	Get effective address
ADD2	36 bit add
NORMAL	Normalize
NEXT	decode instruction
UNPACK	
PACK	
LDRMI	Load RMI
P.FETCH	Institute prefetch
MPY7094	
SMITE	
STMSK	Store under mask
SHFTX	Variable length shift
MPY	18 bit unsigned multiply
DIV	Divide

Reviewing this list and analyzing the outputs of the SMITE compiler, the following items are potential areas for direct microcode:

1. Instruction Decode

   The instruction decode, which determines the opcode and subfields of an instruction, is executed for each emulated instruction; so optimizations that are achieved here may impact the execution speed substantially.

   The emulated program counter and instruction register are manipulated in a manner which can be optimized by new microinstructions to take advantage of the parallelism, use of additional QM-1 hardware (RMI, indexed ALU) and tailored microcode to avoid reloading data. The direct code needs to access data which may be maintained in dedicated QM-1 register. It may be desirable to continue the instruction decode to the point where it performs a multiple branch (CASE) and leaves some of the decode data in QM-1 registers for use by later processors.

2. Operand Fetch

   The operand fetch routine finds the effective address of an instruction, tests the address validity, and returns the value. Since all memory reference instructions use this code, another substantial reduction can be obtained.

   The use of new microinstructions and tailored code should substantially improve execution speed. Since the current SMITE applications have performed the operand fetch in a functional processor, it can be expected that the direct code will use control store and main store variables as the input and return a value in the registers allocated for function values.

3. Special Operators

   There are a number of higher-level functional operations which are translated inefficiently from their SMITE representation.

   A. Normalize

      Normalization is expressed in SMITE by an iterative loop, which shifts the variable until

normalization occurs and increments the counter. The input may be an expression (e.g., X + Y) and possibly a bit position and the output is the normalized value and count.

B. Multiply

Multiplication is expressed in SMITE by an iterative test-and-add-loop. An easier method is to use a multiply microinstruction, such as the one developed for SMITE, to multiply the entire operands or subfields of the operands. The inputs and outputs to this code should be analogous to functional processors.

C. Divide

Division is expressed in SMITE by an iterative loop which tests, shifts, and subtracts data. An additional difficulty is that the dividend is usually longer than the maximum shift width (36 bits) and the SMITE programmer must manipulate data to perform the wide shifts.

A very large number of data storages occur due to this cumbersome code. The division code interface should be analogous to that for functional processors.

D. Long Shifts

A similar problem occurs for certain double register shift operations. The long shift operators operate on two emulated registers and shift them by a specified amount. The register should be SMITE variables, but the amount may be an expression.

E. Floating Point Arithmetic

Floating point operations may obviously be described in terms of fixed point operations on the mantissa and exponent. A series of related manipulations of the mantissa and exponent occurs which may be optimized by new microinstructions (e.g., pack, unpack, floating-add) or by optimized MULTI code.

F. Stack Operations

At the direct code level, emulated stack operations could use push, pull, and block move capabilities in the MULTI instruction set.

G. Decimal Arithmetic

The QM-1 at the nanoprogram level or ALUX instruction can perform decimal arithmetic directly. Direct code to perform decimal arithmetic would operate probably on SMITE variables or functional parameters and return the result in variables or in a register.

H. Ones Complement Arithmetic

Ones complement arithmetic requires an additional level of manipulation and testing on a twos complement machine. A significant improvement could be attained with direct code routines to operate on SMITE variables and return the result in a register

4. Subscripting

The subscript operation in SMITE requires several MULTI operations, especially when the array is packed. A special microinstruction could be created to merge the shifting, addition, and fetch into one instruction. The code, however, would operate on expressions.

5. Direct I/O Access and Tasking

It would be convenient to initiate tasks or I/O operations from the emulator rather than invoking PROD through an external function. For access to QM-1 I/O devices from SMITE or the control of concurrent task, it is more efficient to exercise control from the emulator rather than PROD.

6. Virtual Memory or Associative Memory Actions

There may be special operations, such as SWAP or SEARCH, defined for extensions to the language. Since these operations may involve block data moves or searches, it may be efficient to construct direct code to implement them. The SWAP operation would involve accessing the disk to page memory, which currently can be done only if the emulator returns control to PROD.

Issues to Resolve:

1. Direct Code Directives

The term direct code is somewhat misleading because it implies the programmer codes a routine in the MULTI instruction set and it magically appears in the final object. This approach is insufficient for several reasons:

a. If the direct code specifies actual QM-1 registers, they may not be available at the time (currently being used by the compiler). A method must be created for preserving the integrity of these registers over the direct code.

b. The direct code must access variables in the SMITE data base. These variables may be allocated in main store, control store, or QM-1 registers .

c. The direct code must interface with the surrounding SMITE statements. It may require inputs to be available from preceding statements and supply results to subsequent statements. There must be a method of expressing the parameters of the direct code and their properties so that invocations of the code can be processed by the compiler.

d. The direct code may interface with the control structure. For instance, the direct code may contain a conditional branch and for one branch it may be desirable to invoke a SMITE processor. In other cases, it may be desirable to initiate a CASE operation from the direct code. Although MULTI code can be generated to perform these functions, it violates the structured programming concept.

2. Direct Code Invocation

We must resolve how to process the invocation of direct code and what parameters are necessary. These tasks include the following:

a. The compiler must recognize invocations of direct code.

b. The compiler must be aware of the inputs and outputs which are required by the direct code.

c. The direct code must be checked for violation of certain conditions (such as assembly errors).

3. Compiler Interface

From an implementation standpoint, several issues arise when attempting to interface the direct code with the outputs of the various compiler phases. The direct code must be translated into an analogous representation to parse trees which are generated from the input SMITE computer description. These new trees must contain sufficient information to be processed during the allocation and optimization phases. Additional code generation sequences must be defined to process the direct code, its macro operations and any new microinstructions.

3.2 Direct Code Semantics

Direct code extensions permit the programmer to force the compiler to output specific code sequences for improved efficiency or space utilization. Although the direct code extensions may be regarded as the insertion of specific microprogram sequences in the compiled code, the extensions must contain more information than the MULTI code which is desired by the programmer. The direct code extensions must also interface with the various SMITE compiler phases.

The semantics issue then is to define the format of the direct code extensions. The first task is to review the problems envisioned interfacing the direct code with the SMITE statements. Then the proposed direct code extension format will be discussed. The proposed semantics will then be examined to determine their adequacy.

The direct code must contain information specifying the object code produced and the interface with the SMITE code. The direct code extensions must provide, at least, the following information:

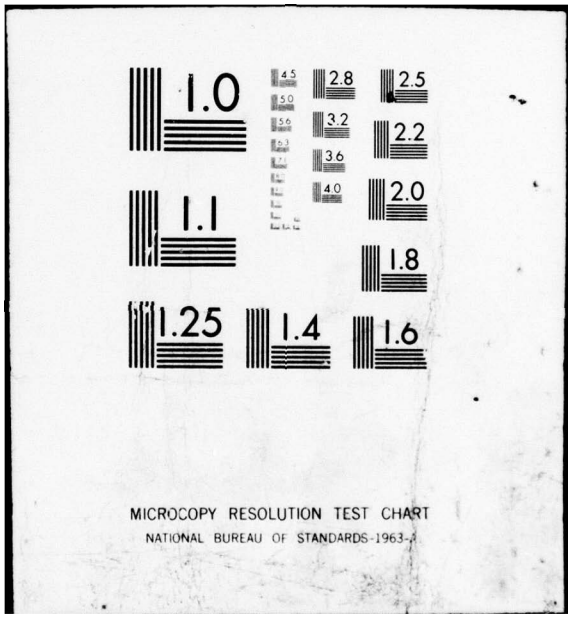
1. The direct code may require parameters at the time of its invocation and may return a value. The number of parameters must be specified along with the width of each.

2. The direct code may address all QM-1 registers. It may also address virtual registers the user declares as SMITE variables. The compiler will assign the virtual registers to actual registers and if necessary, save and restore registers.

AD-A064 910 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF F/G 9/2
SMITE INSTALLATION AND ANALYSIS.(U)
DEC 78 B PRESS, L A PRENTICE F30602-77-C-0089
UNCLASSIFIED TRW-30417-6003-RU-00 RADC-TR-78-212 NL

2 OF 2
AD
A064910

END
DATE
FILMED
4-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

3. The direct code may not directly access variables in the SMITE data base other than those described in 2).

4. Since SMITE is a structured programming language, the direct code must preserve the overall program structure. Furthermore, several phases of the compiler depend upon the structural nature of the program. Enough information must be available to the compiler to check the structure of the direct code and verify that it is irreducible.

5. There are several compiler conventions observed in the code generation of which the programmer should be aware if he wishes to use them. These include the knowledge of the dedicated zero-register, processor call convention, and SMITE stack manipulation. It should be mentioned however, that the user must be very careful in using these sort of items.

The following semantic expressions are allowed for direct code extensions:

1. Representation

A new representation is available for declaring virtual registers (temporaries). The compiler recognizes that the variable will be maintained only in a QM-1 register and that the actual assignment occurs during the code generation phase of the compiler. The temporary variable may be assigned a width. For example the statements

```
DECLARE   TEMP1<35:0>   TEMPORARY
   TEMPZ<17:0>   TEMPORARY;
```

causes a QM-1 register pair and a single QM-1 register to be allocated to the two variables by the compiler. A method of allocating these data items to specific QM-1 registers should also be provided.

The same scope of names rules apply to temporaries. Since the use of temporaries can lead to an unavailability of registers at times, a compiler diagnostic is desirable.

2. Direct Code Definition

To provide for the definition of direct code there are three new keywords to be processed:

DIRECT - specifies the start of a direct code block

SMITE - specifies the end of a direct code block and the return to SMITE statement processing

#<statement> - permits the insertion of one SMITE statement in a direct code block. It is equivalent to SMITE; statement; DIRECT;.

As a sample case to show the use of the definition constructs, assume the target computer is 32 bits wide and a stack push operation is to be coded as a direct code extension:

The prologue may be expressed as follows:

```
. . .
DECLARE   TEMP1   TEMPORARY;
DECLARE   TEMP2   TEMPORARY;
. . .
TEMP1   <- A;
```

The PUSH operation may be expressed in terms of QM-1 MULTI instructions as follows:

```
DIRECT;
   LDD       TEMP2,RZERO,STACK-POINTER      TEMP2=value of
   STACK-POINTER
   ADI       TEMP2,1                        BUMP POINTER
   ST        TEMP2,R.ADR                    NEW
   STACK-POINTER
   SLLI      TEMP2,1
   ADN       R.MX,TEMP2,STACK-2             CURRENT TOP OF
   STACK
   STMSX     TEMP1,1
   STMS      TEMP1+1,R.MX
SMITE;
```

Following are the features to be provided by direct code:

1. MULTI Instructions

   The direct code extensions to the SMITE compiler will be expressed in a language similar to an assembly language for all operations available to the compiler. For new operations additional tabular information is required. (See 2 following.)

   A new data type used for the communication of data

between SMITE statements and direct code must be provided (TEMPORARY data type).

The subset of MULTI instructions used by the SMITE compiler and the microinstructions created for the SMITE compiler are available for direct code. These instructions are expressed in the format used in the MULTI language. The SMITE compiler assumes the burden of:

1. Determining whether the instruction operands are registers, literals, or SMITE variables (TEMPORARY data type).

2. Determining whether transfer labels are defined within the direct code block.

3. Generating operator trees for the MULTI instructions.

4. Checking the syntax of the input instruction for errors. These include an incorrect number of parameters, the wrong type of instruction operands (e.g., a variable in the register field), or illegal characters.

5. Performing tests on the consistency of the operand and the instructions. For instance, a control store operation with a main store variable is an illegal combination.

6. Testing the flowgraph of the direct code to ascertain that it adheres to the structure condition (reducibility) imposed by the compiler.

7. Generating any header and trailer trees needed to alert the compiler of direct code processing which may affect processing.

8. Determining whether SMITE variables used by direct code are declared as TEMPORARY.

2. New MULTI Instruction Definition

Many of the direct code extensions will use new microinstructions created for the particular emulation. Typical examples would be floating point arithmetic, ones-complement arithmetic, and instruction decoding. Nanocode for these operations must be written and added

to the SMITE Application Support System tape. The new nanocode must be made resident in nanostore in some manner.

In the direct code extensions, the new microinstructions will be referenced in the same manner as the standard instructions. This forces the programmer to inform the compiler about the new opcode. A new construct, OPDEF, must be included for all new microinstructions. OPDEF contains the following information:

1. Opcode mnemonic

2. Opcode number

3. Instruction format - numbers and types of operands

The design will require that the various SMITE programmers allocate nanostore so that there are no conflicts.

3. User Defined Decode

A method of allowing the user to define an instruction decoding scheme, other than the CASE statement or cascaded IF statements, must be defined. This method must allow the user to take advantage of the speed afforded by microinstructions when designing the decode mechanism.

3.3 Analysis of Potential Direct Code Applications

The proposed direct code capability should be examined to see if it is adequate for the potential direct code applications stated in the overview.

3.3.1 Decode

The instruction decode will be a new microinstruction. It is up to the user to define what this new micro instruction actually does.

3.3.2 Operand Fetch

The operand fetch instruction uses the outputs of the decode instruction which are maintained in QM-1 registers.

3.3.3 Normalize

A normalize extension can be developed to perform a wide variety of normalizations. Three parameters (value, shift count, bit-position) define the normalization. The bit position specifies the normalization bit. That bit and all bits to the left of it are unchanged by the normalization. The bits to the right are shifted until the normalization bit and next most significant bit differ. The number of bits shifted is returned in the shift count parameter. Nanocode would be developed to perform the normalization.

3.3.4 Multiply

The fixed point 36 bit multiply direct code extension may use the SMITE MPY instruction twice to perform an unsigned multiply.

An 18 bit multiply would be accomplished by a different code sequence which basically would execute one MPY instruction. Variables wider than 36 bits would be multiplied by repeated calls to the 36 bit processor.

The result of the multiply could be placed in contiguous QM-1 registers. SMITE code following the direct code could move the data to the proper locations.

3.3.5 Long Shifts

The direct code for long shifts would benefit substantially from a nanocoded 72 bit shift instruction. The direct codes must be able to use a block of four contiguous QM-1 registers and shift the registers singly or as 36 bit quantities. The following code performs a 72 bit arithmetic right shift.

```
TEMP <- A//Q;
TEMP2 <- SHIFT-COUNT;
DIRECT;
   LOOP:
      LDI     TEMP3,18.          18 BIT AT A TIME
      CPR     TEMP2,TEMP3        SHIFT
      BZS     SIGN,GT18
      MVR     TEMP3,TEMP2        LT 18 BIT COUNT
   GT18:
      SBR     TEMP2,TEMP3        PREPARE FOR NEXT PASS
      MVR     TEMP4,TEMP1+1      SAVE LOWSHIFT BITS
      SHFTX   TEMP1,TEMP2,DBL+RIGHT+ARITH
      SHFTX   TEMP1+2,TEMP2,DBL+RIGHT+LOG
      SBI     TEMP2,18           NUMBER OVFL BITS
      NGR     TEMP2,TEMP2        18-PARTIAL SHIFTCOUNT
```

```
        SHFTX   TEMP4,TEMP2,SNGL+RIGHT+LOG    ISOLATE
        SHFTX   TEMP4,TEMP2,SNGL+LEFT+LOG
        OR      TEMP1+2,TEMP4      OR OVFL OF FIRST SHIFT
        BNZ     TEMP2,LOOP
    SMITE;
```

3.3.6 Decimal Arithmetic

The decimal arithmetic operations will use the QM-1 decimal arithmetic ALU codes, which may be accessed through the ALUX instruction. The decimal coded sum of two 16 bit registers is formed along with the correction word necessary to convert the number back to decimal coded format. The direct code must separate the original variables into 16 bit quantities, perform the QM-1 decimal add, and correct the result.

3.3.7 One's Complement Arithmetic

As an example, ones complement addition may be expressed by the following microcode:

```
    TEMP1 <- SE(A);

    TEMP2 <- SE(B);

    DIRECT;

        ''BEGIN DIRECT CODE''

            ADR     TEMP1,TEMP2          36 bit ADD

            BZS     CARRY,DONE           NO END AROUND CARRY

            ADI     TEMP1,1

        DONE:

        ''HANDLE RESULT = ALL ONES''

    SMITE;
```

3.3.8 Direct I/O or Tasking

The programmer must be responsible for the proper nanocoding of the task control instruction and the interface with the QM-1 system. The interface of this code with the user control system (PROD,EASY etc) needs investigation.

3.4 Conclusion

Direct code extensions will provide a powerful new capability for the SMITE compiler. Most of the problems interfacing direct code with the SMITE expressions are readily solvable.

The process of informing the compiler of new microinstructions poses a compiler design problem. The new operation must specify some form of table-driven logic within the compiler. Any new operation which can be expressed in terms of these options can be passed to the compiler. The failure to provide for the possibility of certain options will limit the type of microinstructions or macros that can be added.

The implementation of the direct code capability presents several compiler problems. The direct code expressions must be converted to compiler trees for later processing. The trees also contain attributes of the expression, which must interface with the SMITE expression attributes. The processing of the direct code requires a true assembler processing of the direct code, which is not present in the current compiler. The processing of direct code also involves generation of flow graphs to determine if the structure is reducible.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.